南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

Lab3-2：基于 UDP 服务设计可靠传输协议并编程实现

冯思程 2112213

年级：2021 级

专业：计算机科学与技术

指导教师：吴英、文静静

2023 年 12 月 1 日

摘要

本次实验，根据要求在实验 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 >1 和接收窗口 =1**，而且我在超时重传的基础上额外实现了快速重传，支持累积确认，并利用到提供的 router 进行包括超时重传机制在内的多种测试，最后综合分析编写报告。

关键字：快速重传、超时重传、滑动窗口、累计确认

目录

一、 预备工作及实验环境

(一) 实验要求与功能

实验要求的基础功能与要求:

1. 实现单向数据传输（一端发数据，一端返回确认）。
2. 对于每个任务要求给出详细的协议设计。
3. 完成给定测试文件的传输，显示传输时间和平均吞吐率。
4. 性能测试指标：吞吐率、延时，给出图形结果并进行分析。
5. 完成详细的实验报告（每个任务完成一份，主要包含自己的协议设计、实现方法、遇到的问题、实验结果，不要抄写太多的背景知识）。
6. 编写的程序应该结构清晰，具有较好的可读性。
7. 提交程序源码、可执行文件和实验报告。
8. 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。

合理的自行扩展功能:

1. 继续使用非阻塞实现本次实验，在非阻塞模式下，套接字操作（sendto 和 recvfrom）不会使程序挂起等待操作完成。这表示程序可以继续执行其他任务，而不是闲置等待，从而提高了整体效率和响应能力。同时在 client 端使用了多线程编程提升效率。
2. 在断开连接的设计中，考虑到了最后一个 ACK 可能丢失的情况，让 client 等待 2MSL 来确保 client 和 server 的正确关闭。（延续 3-1 的设计）
3. 在设计协议的时候考虑到了多种特殊情况，在分析的时候也进行了丰富的测试对比分析。
4. 实现了 TCP 协议中的快速重传机制。

(二) 实验环境与说明

具体的实验环境配置如下:

Windows 版本	vs code 版本	docker 版本	g++ 版本
windows11	1.82.0	24.0.6	8.1.0

表 1: 实验环境说明表

代码文件使用 GBK 编码，以支持中文字符。makefile 文件使用正常的 UTF-8 编码。
感谢老师与助教的审查批阅与指正，辛苦!

二、 实验过程

(一) 协议设计

下面分点进行说明:

1. 报文格式

报文格式沿用 3-1 实验中的设计，如下：报文分为两个部分：报文头和报文段。

报文头 报文头一共包括源 IP（4 字节）、目的 IP（4 字节）、源 PORT（2 字节）、目的 PORT（2 字节）、序列号（4 字节）、确认号（4 字节）、发送文件大小（4 字节）、标志（2 字节）、校验和（2 字节）。

一共 28 字节，其中由于本次实验测试使用的是本地回环地址，所以源 IP 和目的 IP 这两个字段一直没用上。其中发送文件大小是记录要发送的文件的大小的，后文会进一步讲到这个字段。

报文段 这里我设计的报文段就是装载数据的数据段。这里最大字节数我设置为 10000 字节，因为 router 转发的数据包最大是 15000 字节。

源 IP	
目的 IP	
源 PORT	目的 PORT
序列号	
确认号	
发送文件大小	
标志	校验和
报文段	

表 2: 报文格式

标志位 报文头的标志字段可以从下面设计的四种选择若干种进行加到 flag 上从而实现标志设置，其中包括 TCP 协议中用到的 SYN、ACK、FIN，还有一个我自己设计的 SFileName 标志，这个是用来表示传输文件的具体数据前发送文件名和文件大小的那条报文的，代码如下：

标志位

```
//设置不同的标志位，用到的包括SYN、ACK、FIN，SFileName表示传输了文件名字
const unsigned short SYN = 0x1;//0001
const unsigned short ACK = 0x2;//0010
const unsigned short FIN = 0x4;//0100
const unsigned short SFileName = 0x8;//1000
```

2. 建立连接：三次握手（包含超时重传）

三次握手的设计也沿用 3-1 实验中的设计，如下：三次握手基于 TCP 协议进行设计的，实现的示意图如下：

图 1: 三次握手示意图

这里在 UDP 的不可靠信道上进行三次握手来建立一个连接，需要 client 和 server 总共发送 3 个包。三次握手的目的是连接服务器指定端口，建立连接，并同步连接双方的序列号和确认号。这里的序列号我设计的是双端（client 和 server）各自维护各自序列号，然后确认号的维护机制是根据情况相应地设置。

第一次握手 client 将报文标志位 SYN 置为 1，随机产生一个序号值 seq=x。

第二次握手 server 接收到了 client 发起的连接请求后进行回应，这条报文 ACK、SYN 置 1，确认号是 x+1（client 的发来消息的序列号 +1）同时，自身的序列号也会随机出一个 y。

第三次握手 client 发给 server 的一个确认报文，序列号是 x+1（这也是唯一和 TCP 协议不同的地方，ACK 报文在 TCP 协议中是不消耗序列号的，这里我将其设置也会消耗序列号），确认号是 server 发来的报文的序列号 +1。

由于这里存储的是相对序列号，所以图中的 x 和 y 可以都看作 0。

三次握手的超时重传机制 这里我设计了基于 rdt3.0 协议的超时重传机制，下面分别是 client 和 server 的重传机制实现流程：

client：

1. 发送第一次握手，在发送后接着就开始计时
2. 接收 server 发来的第二次握手，在这个不断等待接收的过程中，如果第一次握手超时，则会重传第一次握手并重新开始计时。这里注意，我设置最大重传次数为 10 次，如果 10 次重传都失败则会退出。
3. 发送第三次握手，这是一个 ACK 报文，不需要进行重传。发送后 client 成功建立连接。

server：

1. 接收第一次握手
2. 接收到第一次握手后，发送第二次握手消息并在发送后接着进行计时。
3. 接收 client 的第三次握手，在这个不断等待接收的过程中，如果第二次握手超时，则会重传第二次握手并重新开始计时。这里注意，我设置最大重传次数为 10 次，如果 10 次重传都失败则会退出。接收第三次握手成功后 server 成功建立连接，现在双方可以开始进行数据传输了。

注意这里握手完成后需要在 clinet 端对滑动窗口的 start 和 arrive 进行更新，因为我本次实验采用的也是双端各自维护 seq 值，而基于我的设计握手 client 端的握手一共占用了两个序列号（第一次握手和第三次握手），所以 client 端的序列号之后是从 2 开始的。

3. 不可靠信道上的可靠数据传输：差错检验、超时重传、三次快速重传、流量控制（滑动窗口）、累计确认

本次实验中从 client 向 server 发送文件，文件较大，需要拆分成多个数据报文进行传输，一次完整的文件传输的数据报文一共分为三类，如下：（与 3-1 实验一致）

1. 第一类数据报文是装载文件名字和文件大小的数据报文。
2. 第二类数据报文是满载数据报文，这类报文传输的是具体的文件数据，满载就是这个报文的报文段可以装满。
3. 第三类数据报文是未满载数据报文，这类报文只能是 0 个或者 1 个，装载的数据大小是文件大小除以报文段最大装载大小的余数。

数据传输中从之前的停等机制改进为基于 GBN 协议的滑动窗口机制，这里的滑动窗口其实可以理解为“流水线”，在原来的停等机制中，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，这样会导致较长的时延。所以这里进行了改进，在窗口范围内，可以在没有收到前面发送的数据报文的确认报文继续发送，直到窗口被塞满。对于序列号来说，我依旧沿用 3-1 实验中中由 client 和 server 双端各自维护序列号进行**递增 1** 的方法。差错检验与 3-1 实验一致。在 server 端继续沿用 3-1 实验中 ACK 报文也会占用序列号的设计。接下来我具体的讲解一下**滑动窗口**与**累计确认**的设计。

首先展示一下时序图：

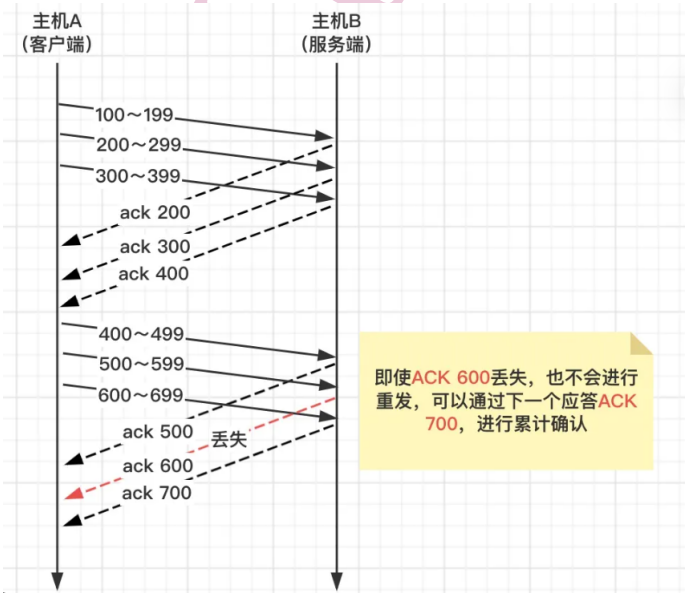

图 2: 滑动窗口与累计确认时序图

滑动窗口 对于 client 端，我这里设计了固定滑动窗口大小的协议，首先根据下图来讲解一下滑动窗口的含义，client 端的数据报文可以分为四个部分，绿色部分代表报文发送了而且收到了 ACK 报文，黄色部分代表报文发送了但是还没收到 ACK 报文，蓝色部分代表未发送但是窗口

还可以发送的报文，灰色部分代表未发送而且超窗口大小限制的报文。这里设计是窗口大小固定并不会双端商议变换窗口大小 CWND，这里我初始设定为 10。然后我设计了两个整型标志代表当前 client 端窗口的起始 seq 值和已经发送到的 seq 值，这两个可以体现窗口的使用情况，arrive-start 就是窗口已发送未确认的报文数量，CWND-（arrive-start）就是还可以发送的报文数量。简单来说，start 就是已发送未确认的第一个报文的 seq 值，arrive 是下一次要发送报文的 seq 值。这样也可以方便我维护序列号。

同时我这里实现了一个基于队列的缓冲区，用于按序存放已发送未确认的报文（即窗口内的这些已发送未确认的数据报文）这是为了后文的重传机制的实现。

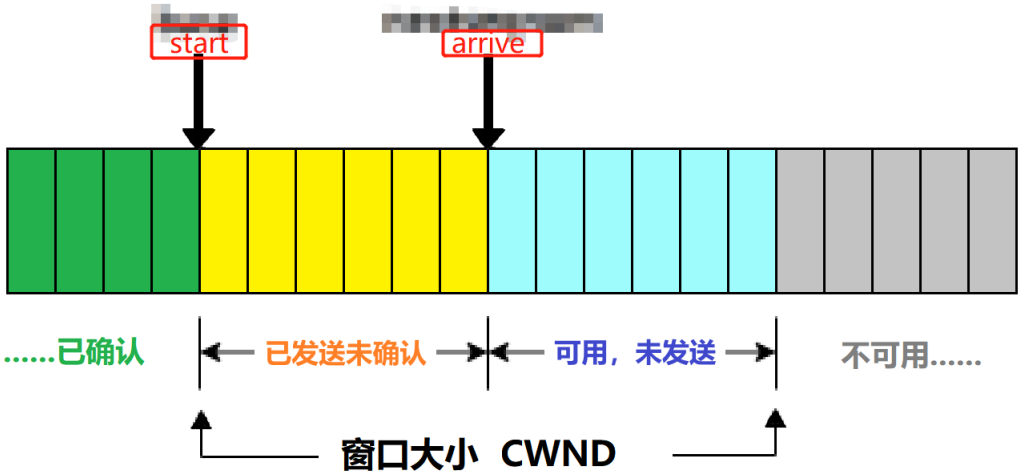

图 3: 滑动窗口

超时重传与快速重传 这是两种不同的机制，超时重传机制是类似 3-1 实验中的设计（有所不同的是，这次不会有最大重传次数了，而是可以无限重传），接下来仔细讲解一下快速重传：

快速重传机制不以时间为驱动，而是以数据驱动重传。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。所以这里我设计为如果接收到三次相同的重复 ACK 报文，则认为是报文丢失，而且这里我会认为窗口内已发送未确认的报文都有可能会丢失，于是我会重传所有已发送未确认的报文。

超时重传也会重传所有已发送未确认的报文。

这里重传会利用上文提到的缓冲区进行实现，会遍历缓冲区这个队列，按序进行重传，重传一遍后缓冲区队列与原来保持一致。

累计确认 这种机制用于确认接收到的数据报文，并告知发送端哪些数据报文已被成功接收。累计确认机制其实主要是应对发送的 ACK 报文丢失的问题，这里我设计接收方窗口大小为 1，确保接收数据报文是**按序接收**，累计确认保证只要发送端收到了一个 ACK 报文，就说明前面的数据报文都被正确接收到了。如果发生了 ACK 报文丢失，我仍可以通过后面发送的 ACK 报文去一起确认之前收到过所有的报文，但是需要注意的是我们本次实验的 server 端的窗口大小是 1，即按序接收。

不同情况分析 接下来面对理想、丢包、延时、失序等情况，具体地来分析我的协议设计：

1. 第一种情况就是在**理想情况**下，没有丢包、没有延时、没有失序问题。初始时，client 端的

窗口的 start 和 arrive 都是 2（由于握手消耗了两个序列号），然后开始发送数据报文，每发送一个 arrive 递增，start 不变，直到窗口大小用尽。然后 server 会按顺序接收 client 发来的数据包，每接收到一个先检查序列号，如果是正确的序列号则回复一个 ACK 包，确认号是发来数据包的序列号，ACK 包的序列号是 server 维护的序列号自动递增，同时会接收这个数据报文，交给上层应用。然后 client 端接收到正确的确认报文，则窗口右移，同时更新缓冲区（将小于等于确认报文确认号的报文都踢掉）窗口移动后，窗口又有空闲位置可以发送报文了，就继续发送报文，不断重复上述过程直到所有报文都发送完成。

2. 丢包和失序和超时问题在这里可以一起进行说明，因为可以一起进行处理。如果 client 丢失数据包会触发快速重传机制，因为 server 端会收到不等于期待序列号的数据报文，然后其会一直返回同一个 ACK 报文（并丢弃这个数据报文，与 3-1 实验一致），ACK 报文丢失在上文累计确认已经说到了，可以通过后面发送的 ACK 报文一起确认（但实际上 server 发送的 ACK 报文在本次实验中由于 router 的设置并不会丢失）。对于超时问题处理类似 3-1 实验，会进行重传，也会将缓冲区中的所有数据报文都重传，同时没有最大重传次数和快速重传一样可以无限重传。对于失序问题，接收端也会收到不等于期待序列号的数据报文，这也导致 server 回复同一个 ACK 报文（并丢弃这个数据报文）。对于 client 端如果收到 ACK 报文小于 start 说明是一个失序报文，直接忽略。client 只有在收到大于等于 start 的 ACK 报文才会更新窗口（这代表是一个正确的 ACK 报文）

最后我展示一下数据传输过程中的 client（发送端）和 server（接收端）的状态机，具体如下：

client(发送端)：

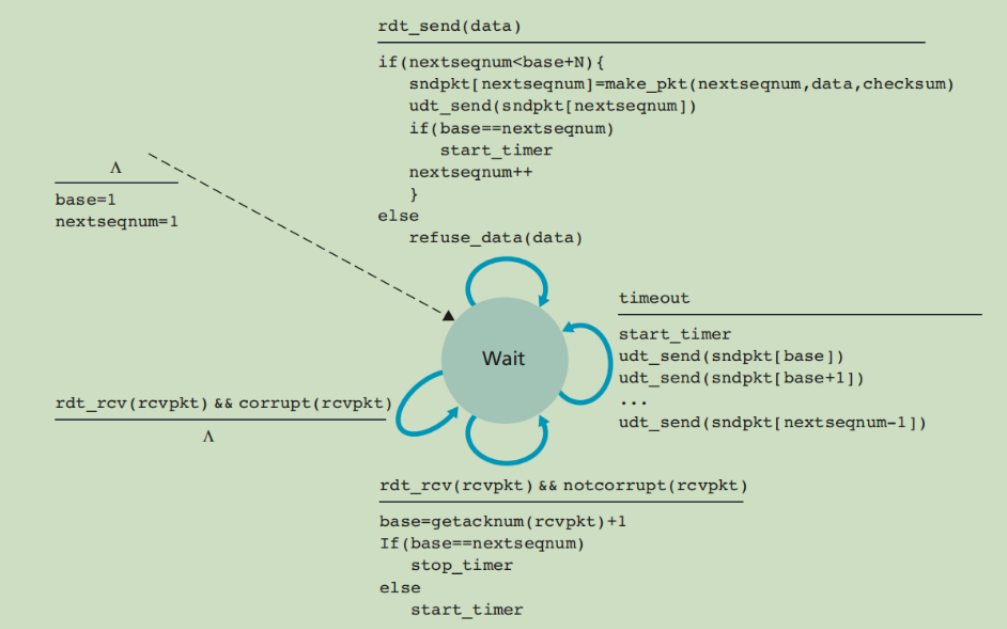

图 4: GBN 发送端状态机

server(接收端)：

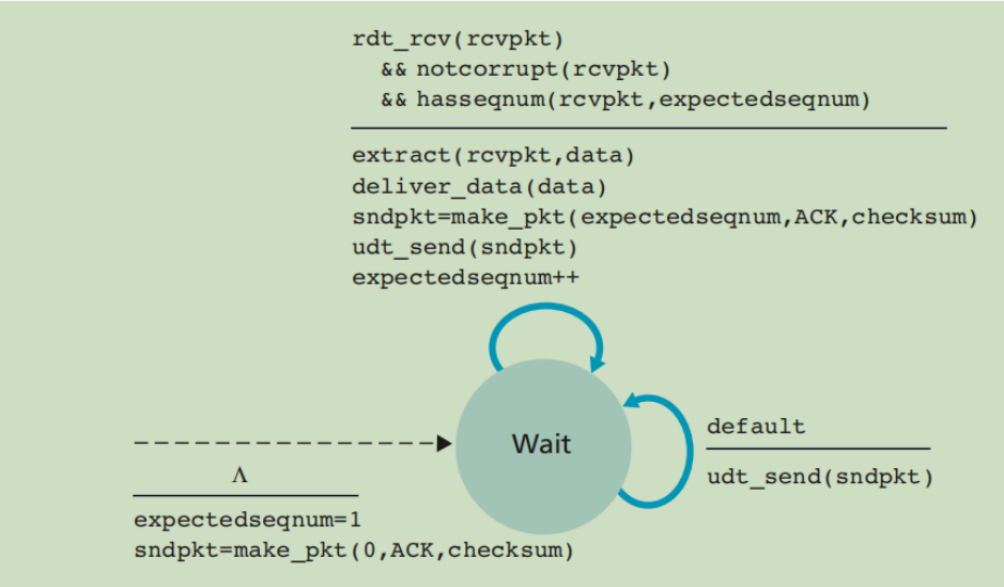

图 5: GBN 接收端状态机

4. 断开连接：四次挥手（包含超时重传）

四次挥手的设计也沿用 3-1 实验中的设计，如下：我是基于 TCP 协议进行设计的，实现的示意图如下：

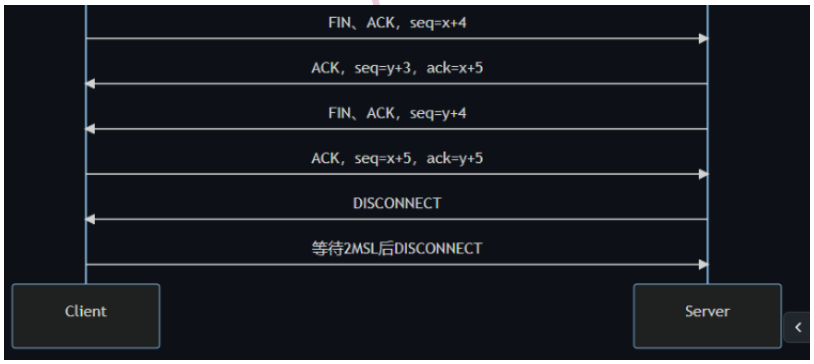

图 6: 四次挥手

第一次挥手 数据传输完成后，先由 client 向 server 发送一个将 FIN、ACK 置 1 的报文，代表想进行断开连接，这个报文的序列号是从前面的数据报文序列号进行递增的，注意我的设计种这个条报文不设置确认号。

第二次挥手 server 收到 client 发来的第一次挥手报文后，需要回复一个 ACK 报文，这个报文我这里也会占据一个序列号（这里我设计的协议 ACK 报文都是占据序列号的，这一点与 TCP 有所区别）序列号就是在 server 维护的序列号自动向下递增 1。确认号是 client 发来报文的序列号 +1，这一点与 TCP 相同。

第三次挥手 server 完成所有的数据传输后（本次实验中在挥手前就已经完全了全部的数据传输，因为这里使用的是停等机制），server 会向 client 发送一个报文表示也可以断开连接，这条

报文将 FIN、ACK 置 1，序列号还是 server 维护的序列号自动向下递增 1，这条报文不设置确认号。

第四次挥手 client 收到 server 发来的第三次挥手后，会回复一个 ACK 报文，这个报文的序列号是 client 维护的序列号自动向下递增 1（这里我设计的协议 ACK 报文都是占据序列号的，这一点与 TCP 有所区别）。确认号是 server 发来的第三次挥手报文的序列号 +1。

四次挥手的超时重传机制 这里我设计了基于 rdt3.0 协议的超时重传机制，下面分别是 client 和 server 的重传机制实现流程：

client：

1. 发送第一次挥手，在发送后接着就开始计时
2. 接收 server 发来的第二次挥手，在这个不断等待接收的过程中，如果第一次挥手超时，则会重传第一次挥手并重新开始计时。这里注意，我设置最大重传次数为 10 次，如果 10 次重传都失败则会退出。
3. 接收第三次挥手
4. 发送第四次挥手，这是一个 ACK 报文
5. 等待 2MSL，由于 client 端并不知道 server 端是否正确收到了最后一次 ACK 报文，为了防止最后一个 ACK 报文丢失，要等待 2MSL，如果再次收到了消息，则说明最后一个 ACK 丢失，会重传第四次挥手的 ACK 包，注意这里是只重传了一次，没有设置最大重传次数。等待结束后则会退出。

server：

1. 接收第一次挥手
2. 接收到第一次挥手后，发送第二次挥手，这是一个 ACK 报文。
3. 发送第三次挥手，并开始计时。
4. 接收 client 的第四次握手，在这个不断等待接收的过程中，如果第三次挥手超时，则会重传第三次挥手并重新计时。这里注意，我设置最大重传次数为 10 次，如果 10 次重传都失败则会退出。接收到第四次挥手就会退出。

5. 日志输出

输出了三次握手和四次挥手的完整过程，其中输出了发送的报文的几乎所有字段，包括但不限于序列号、确认号、标志等。

在数据传输过程中，首先在 client 端将发送的数据报文的所有字段进行输出，而且在 server 端会输出要传输的文件名字和大小，并会记录收到的数据报文。这里对上次的输出进行了简化，减少了很多没有必要的输出。

对于重传机制的实现，在全过程，我也编写了完整的输出，可以在后文的测试中观察到，在本次实验中也加入了一些可以体现累计确认和快速重传的输出。

对于可能出现的报错情况，我也都进行了完整的报错处理。

在 client 端文件传输结束后输出了总体传输时间、平均吞吐率。

(二) 核心代码实现

1. 报文格式核心代码与校验和

报文的实现是通过结构体进行实现的，用对应的字节长度定义上文协议设计中说明的不同字段，代码与 3-1 实验一致。

报文格式

```
struct Message
{
        //省略其余与3-1实验相同代码
};
```

同时这里实现了设置校验和和检验校验和的函数，校验和计算遵循互联网校验和的标准计算方法。

设置校验和这里采用的粒度是 2 字节（16 位）。首先将校验和字段清零，然后对结构体执行端进位加法（端进位加法就是高 16 位不为 0 的话，就将其加到低 16 位，不断重复直到高 16 位为 0）。最后将计算得到的 32 位和的低 16 位取反，存储为校验和。这确保了在传输或存储过程中数据包的完整性可以被接收方验证。

设置校验和

```
void Message::setCheck()
{
        //省略其余与3-1实验相同代码
}
```

检验校验和这里采用的粒度也是 2 字节（16 位），对整个结构体进行端进位加法，来验证数据的完整性。如果在将所有 16 位加起来之后，计算得到的 32 位和的低 16 位全为 1，则说明校验和是正确的，数据在传输或存储过程中未发生变化，返回 true。如果低 16 位不全为 1，则校验和验证失败，返回 false。

检验校验和

```
bool Message::check()
{
        //省略其余与3-1实验相同代码
        return false;

}
```

2. 三次握手核心代码

这里具体的协议设计，包括连接协议和超时重传协议已经在上文的协议设计中进行了具体的说明，下面讲一下实现：这里的实现流程与协议设计中说明的一样，在连接的时候必须是一端等待另一端的消息，也就是流量控制使用停等协议，收发报文利用的函数主要是 UDP 中常用的 **sendto 函数**和 **recvform 函数**。注意这里在 client 端的握手函数中与 3-1 实验中有一处变化，就是在最后为了维护 seq 值对滑动窗口的 start 和 arrive 进行**更新**。

client 端 ：

client 三次握手

```
//实现client的三次握手
bool threewayhandshake(SOCKET clientSocket, SOCKADDR_IN serverAddr)
{
        //省略其余与3-1实验相同代码
        start=initrelseq+1;
        arrive=initrelseq+1;
        return true;
}
```

server 端 ：

server 三次握手

```
//实现server的三次握手
bool threewayhandshake(SOCKET serverSocket, SOCKADDR_IN clientAddr)
{
        //省略其余与3-1实验相同代码
        return false;
}
```

3. 可靠数据传输核心代码

这里的实现流程与协议设计中的一致，具体代码如下:

client 端 ：

首先展示一下进行文件发送的主体函数，首先会将要发送的文件读成字节流，然后开始发送数据报文，这里接收 ACK 报文我另外开辟了一个线程进行接收，现在 client 端有两个线程一个是主线程即发送数据报文，另外一个是 ACK 报文接收线程。两个线程会共享所需的变量如下:（注释给出了解释，后文也有用法说明）

client 双线程共享变量定义

```
// 定义窗口大小为10
#define windowssize 10

//缓冲区存已发送未确认的message
queue<Message> messageBuffer;

//辅助加锁
HANDLE mutex;
//client端维护的序列号
int initrelseq = 0;
//滑动窗口的开始和到达
int start = 0;
int arrive = 0;
//计时
```

```
int messagestart;
//标志位
bool finish = false;
bool resend = false;
```

然后开始发送数据报文，如果窗口未塞满，则可以继续发送，同时会递增 arrive。当超时或者触发快速重传的时候，会利用基于队列的缓冲区进行遍历重传，这里会将缓冲区中的所有消息进行重传，同时每个数据报文重传后会重新加到队尾，这样如果重传一直不成功可以实现无限次数重传，这里重传后会重新计时。而且这里主线程和 ACK 报文接收线程是并行的，所以可能会发生共享变量修改发生错乱的情况，为了处理这种情况，我这里使用了互斥锁来实现输出的原子性保证，具体代码如下：

client 主线程：数据传输

```
//实现文件传输
void sendFileToServer(string filename, SOCKADDR_IN serverAddr, SOCKET
    clientSocket)
{
        mutex = CreateMutex(NULL, FALSE, NULL); // 创建互斥锁
        int starttime = clock();
        string realname = filename;
        filename="测试文件\\"+filename;
        //将文件读成字节流
        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;//可以装满的报文数量
        int leftNum = fileSize % MaxMsgSize;//剩余数据量大小
        int nummessage;
    //算出一共要发送的数据报文数量
        if(leftNum!=0){
                nummessage=batchNum+2;
        }
        else{
                nummessage=batchNum+1;
```

```
35          }
36
37          parameters useparameter;
38          useparameter.serverAddr = serverAddr;
39          useparameter.clientSocket = clientSocket;
40          useparameter.nummessage = nummessage;
41          HANDLE hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
                recvackthread, &useparameter, 0, 0);
42
43          int count=0;
44      while(1){
45                  if (arrive < start + windowssize && arrive < nummessage + 2){
46                          Message datamessage;
47                          if (arrive == 2){
48                                  //发送文件名和文件大小，文件大小放在头部的
                                        size字段了，设置了SFileName标志位
                                        datamessage
49                                  datamessage.SrcPort = ClientPORT;
50                                  datamessage.DestPort = RouterPORT;
51                                  datamessage.size = fileSize;
52                                  datamessage.flag += SFileName;
53                                  datamessage.SeqNum = arrive;
54                                  //将文件名放在前面并加一个结束符
55                                  for (int i = 0; i < realname.size(); i++)
56                                          datamessage.data[i] = realname[i];
57                                  datamessage.data[realname.size()] = '\0';
58                                  datamessage.setCheck();
59                          }
60                          else if (arrive == batchNum + 3 && leftNum > 0){
61                                  //未满载的数据报文段发送，如果是整除就不发送
                                        这个报文了
62                                  datamessage.SrcPort = ClientPORT;
63                                  datamessage.DestPort = RouterPORT;
64                                  datamessage.SeqNum = arrive;
65                                  for (int j = 0; j < leftNum; j++)
66                                  {
67                                          datamessage.data[j] = fileBuffer[
                                                batchNum * MaxMsgSize + j];
68                                  }
69                                  datamessage.setCheck();
70                          }
71                          else{
72                                  //满载的数据报文段发送，类似批量发送，这里为
                                        了简便并没有设置标志位
73                                  //发送的数据报文并不需要设置标志位，这里是为
                                        了方便快捷
74                                  datamessage.SrcPort = ClientPORT;
75                                  datamessage.DestPort = RouterPORT;
```

```
76                          datamessage.SeqNum = arrive;
77                          for (int j = 0; j < MaxMsgSize; j++)
78                          {
79                                  datamessage.data[j] = fileBuffer[
                                        count * MaxMsgSize + j];
80                          }
81                          datamessage.setCheck();
82                          count++;
83                  }
84                  if (start == arrive)
85                  {
86                          messagestart = clock();
87                  }
88                  {
89                  WaitForSingleObject(mutex, INFINITE);   // 等待并获取
                        互斥锁
90                  //存到缓冲区中
91                  messageBuffer.push(datamessage);
92                  sendto(clientSocket, (char*)&datamessage, sizeof(
                        datamessage), 0, (sockaddr*)&serverAddr, sizeof(
                        SOCKADDR_IN));
93                  arrive++;
94                  cout << "client发送:␣seq␣=␣" << datamessage.SeqNum <<
                        ",␣校验和␣=␣" << datamessage.checkNum<<"的数据报
                        文"<<endl;
95                  cout <<"[窗口情况(发送消息后瞬间)]␣窗口大小为"<<
                        windowssize<<"已发送但未收到确认的报文数量为"<<(
                        arrive-start)<<endl<<endl;
96                  ReleaseMutex(mutex);     // 释放互斥锁
97                  }
98          }
99
100         // 超时重传+快速重传
101         if (resend   || clock() - messagestart > MAX_WAIT_TIME)
102         {
103                 if(resend){
104                         cout<<"三次相同冗余ACK报文触发快速重传机制"<<
                                endl;
105                 }
106                 {
107                 WaitForSingleObject(mutex, INFINITE);   // 等待并获取
                        互斥锁
108                 // 从缓冲区中重传消息
109                 for (int i = 0; i < arrive-start; i++) {
110                         Message resendMsg = messageBuffer.front();
111
112                         sendto(clientSocket, (char*)&resendMsg,
                                sizeof(resendMsg), 0, (sockaddr*)&
```

```
                                    serverAddr, sizeof(SOCKADDR_IN));
                                cout<<"正在重传:␣seq␣=␣"<<resendMsg.SeqNum<<"
                                    的数据报文"<<endl;

                                // 将消息重新放入队尾，以便后续可能的再次重传
                                messageBuffer.push(resendMsg);
                                messageBuffer.pop();
                            }
                            ReleaseMutex(mutex);     // 释放互斥锁
                            }
                            messagestart = clock();
                            resend = false;
                        }

                        //如果结束就退出
                        if(finish == true){
                                break;
                        }
                }
                CloseHandle(hThread);

                //计算传输时间和吞吐率
                int endtime = clock();
                cout << "\n总传输时间为:" << (endtime - starttime) << "ms" << endl;
                cout << "平均吞吐率:" << ((float)fileSize) / (endtime - starttime)
                    << "bytes/ms" << endl << endl;
                delete[] fileBuffer;//释放内存
                CloseHandle(mutex); // 清理互斥锁
                return;
}
```

下面是新创建的 ACK 报文接收线程，接收到大于等于 start 的 ACK 报文则会更新 start，同时会将缓冲区更新，利用队列中序列号有序的特性，弹出所有序列号小于等于这个 ACK 报文确认号的数据报文。如果小于 start 则不更新。当接收到最后一个 ACK 报文后会设置 finish 标志位并返回。这里还实现了触发三次重复 ACK 报文的快速重传机制，这里用 resend 来进行表示是否触发了快速重传机制（用一个 errorack 记录上一次 ACK 报文的确认号，用 errorcount 记录重复 ACK 报文的次数，如果超过三次就设置 resend 为 true）。而且这里主线程和 ACK 报文接收线程是并行的，所以可能会发生共享变量修改发生错乱的情况，为了处理这种情况，我这里使用了互斥锁来实现输出的原子性保证，具体代码如下：

先展示一下更新缓冲区的函数，如下：

更新缓冲区函数

```
//缓冲区更新，收到了ACK报文，则需要把缓冲区那些seq值小于等于ACK报文的ack值的
    那些message弹出队列
void updateBuffer(int ackNum) {
    while (!messageBuffer.empty()) {
        const Message& frontMsg = messageBuffer.front();
        if (frontMsg.SeqNum <= ackNum) {
```

```
            messageBuffer.pop();  // 删除已确认的消息
        } else {
            break;  // 一旦遇到一个未确认的消息，停止循环
        }
    }
}
```

然后展示接收 ACK 报文线程的函数，如下：

接收 ACK 报文线程

```
//接收ack的线程
DWORD WINAPI recvackthread(PVOID useparameter)
{
        mutex = CreateMutex(NULL, FALSE, NULL); // 创建互斥锁
        parameters* p = (parameters*)useparameter;
        SOCKADDR_IN serverAddr = p->serverAddr;
        SOCKET clientSocket = p->clientSocket;
        int nummessage = p->nummessage;
        int AddrLen = sizeof(serverAddr);

        int errorack = -1;
        int errorcount = 0;

        unsigned long mode = 1;
        ioctlsocket(clientSocket, FIONBIO, &mode);
        while (1)
        {
                Message recvMsg;
                int recvByte = recvfrom(clientSocket, (char*)&recvMsg, sizeof
                    (recvMsg), 0, (sockaddr*)&serverAddr, &AddrLen);
                //成功收到消息
                if (recvByte > 0)
                {
                        //检查校验和
                        if (recvMsg.check())
                        {
                                if (recvMsg.AckNum >= start){
                                        {
                                        WaitForSingleObject(mutex, INFINITE);
                                            // 等待并获取互斥锁
                                        updateBuffer(recvMsg.AckNum);
                                        start = recvMsg.AckNum + 1;

                                        cout << "client收到:ack=" <<
                                            recvMsg.AckNum << "的ACK报文" <<
                                            endl;
                                        cout <<"[窗口情况（接收到消息后瞬间）
                                            ]窗口大小为"<<windowssize<<"已发
                                            送但未收到确认的报文数量为"<<(
```

```
                                        arrive-start)<<endl<<endl;
33                                      ReleaseMutex(mutex);
                                                        // 释放互斥锁
34                                      }
35                                  }
36                                  if (start != arrive){
37                                      messagestart = clock();
38                                  }
39                                  //判断结束的情况
40                                  if (recvMsg.AckNum == nummessage + 1)
41                                  {
42                                      cout << "\n文件传输结束" << endl;
43                                      finish = true;
44                                      return 0;
45                                  }
46                                  //连续收到三个相同的错误ACK触发快速重传
47                                  //快速重传，基于TCP的快速重传算法实现，减少超
                                        时重传的等待时间
48                                  if (errorack != recvMsg.AckNum)
49                                  {
50                                      errorcount = 0;
51                                      errorack = recvMsg.AckNum;
52                                  }
53                                  else
54                                  {
55                                      errorcount++;
56                                  }
57                                  if (errorcount == 3)
58                                  {
59                                      resend = true;
60                                  }
61
62                              }
63                              //若校验失败不对，忽略并继续等待
64                          }
65                  }
66                  CloseHandle(mutex); // 清理互斥锁
67                  return 0;
68  }
```

server 端 ：

server 端会按序进行报文的接收,如果发来的报文序列号是正确的序列号则会回复一个 ACK 报文,确认号等于这个数据报文的序列号。否则会丢弃这个数据报文并回复一个 server 端累计确认的最后一个报文号,这里 server 的窗口是 1,所以就是正确序列号-1,接收函数还是 **recvMessage 函数**，server 端修改很少，主要是接收数据报文如果没有接收到正确序列号的数据报文的处理与之前有所不同，具体代码如下:

server 数据接收

```
1
2  //实现文件接收
3  void recvFileFormClient(SOCKET serverSocket, SOCKADDR_IN clientAddr)
4  {
5          int AddrLen = sizeof(clientAddr);
6          //接收文件名和文件大小
7          Message nameMessage;
8          unsigned int fileSize;
9          char fileName[40] = {0};
10         while (1)
11         {
12                 int recvByte = recvfrom(serverSocket, (char*)&nameMessage,
                       sizeof(nameMessage), 0, (sockaddr*)&clientAddr, &AddrLen)
                       ;
13                 if (recvByte > 0)
14                 {
15                         //如果成功收到装载文件名和文件大小的消息，则进行一下
                           输出并回复对应的ACK报文
16                         //这里我设计的是基于rdt3.0协议，将ack值设为发来报文的
                           seq值
17                         if (nameMessage.check() && (nameMessage.SeqNum ==
                           initrelseq + 1) && (nameMessage.flag & SFileName)
                           )
18                         {
19                                 fileSize = nameMessage.size;
20                                 for (int i = 0; nameMessage.data[i]; i++)
21                                         fileName[i] = nameMessage.data[i];
22                                 cout << "\n接收文件名: " << fileName << ", 文
                                   件大小: " << fileSize << endl<<endl;
23
24                                 Message replyMessage;
25                                 replyMessage.SrcPort = ServerPORT;
26                                 replyMessage.DestPort = RouterPORT;
27                                 replyMessage.flag += ACK;
28                                 replyMessage.SeqNum=initrelseq++;
29                                 replyMessage.AckNum = nameMessage.SeqNum;
30                                 replyMessage.setCheck();
31                                 sendto(serverSocket, (char*)&replyMessage,
                                       sizeof(replyMessage), 0, (sockaddr*)&
                                       clientAddr, sizeof(SOCKADDR_IN));
32                                 cout << "server收到: seq␣=␣" << nameMessage.
                                       SeqNum << "的数据报文"<<endl;
33                                 cout << "server发送: seq␣=␣"<<replyMessage.
                                       SeqNum<<", ack␣=␣" << replyMessage.AckNum
                                        << "␣的ACK报文, 检验和是" <<replyMessage
                                       .checkNum<< endl<<"装载文件名和文件大小的
                                       报文接收成功, 下面开始正式传送数据段"<<
```

```
                                endl<< endl;
34                          break;
35                      }
36
37                      //如果seq值不正确，则丢弃报文，返回累计确认的ACK报文
                            (ack=initrelseq-1)
38                      else if (nameMessage.check() && (nameMessage.SeqNum
                            != initrelseq + 1) && (nameMessage.flag &
                            SFileName))
39                      {
40                          Message replyMessage;
41                          replyMessage.SrcPort = ServerPORT;
42                          replyMessage.DestPort = RouterPORT;
43                          replyMessage.flag += ACK;
44                          replyMessage.SeqNum=initrelseq+1;
45                          replyMessage.AckNum = initrelseq;
46                          replyMessage.setCheck();
47                          sendto(serverSocket, (char*)&replyMessage,
                                sizeof(replyMessage), 0, (sockaddr*)&
                                clientAddr, sizeof(SOCKADDR_IN));
48                          cout << "[累计确认(错误seq值)]server收到␣
                                seq␣=␣" << nameMessage.SeqNum << "的数据
                                报文，并发送␣ack␣=␣" << replyMessage.
                                AckNum << "的ACK报文" << endl;
49                      }
50                  }
51              }
52
53
54              int batchNum = fileSize / MaxMsgSize;//可以装满的报文数量
55              int leftNum = fileSize % MaxMsgSize;//剩余数据量大小
56              BYTE* fileBuffer = new BYTE[fileSize];
57
58              //满载的数据报文段接收
59              for (int i = 0; i < batchNum; i++)
60              {
61                  Message dataMsg;
62                  if (recvMessage(dataMsg, serverSocket, clientAddr))
63                  {
64                      cout << "第" << i+1 << "个满载数据报文接收成功" <<
                            endl<< endl;
65                  }
66                  else
67                  {
68                      cout << "第" << i+1 << "个满载数据报文接收失败" <<
                            endl<< endl;
69                      return;
70                  }
```

```
71                  //读取数据
72                  for (int j = 0; j < MaxMsgSize; j++)
73                  {
74                          fileBuffer[i * MaxMsgSize + j] = dataMsg.data[j];
75                  }
76          }
77
78          //未满载的数据报文段接收, 如果是整除就不接收这个报文了
79          if (leftNum > 0)
80          {
81                  Message dataMsg;
82                  if (recvMessage(dataMsg, serverSocket, clientAddr))
83                  {
84                          cout << "未满载的数据报文接收成功" << endl<< endl;
85                  }
86                  else
87                  {
88                          cout << "未满载的数据报文接收失败" << endl<< endl;
89                          return;
90                  }
91                  //读取数据
92                  for (int j = 0; j < leftNum; j++)
93                  {
94                          fileBuffer[batchNum * MaxMsgSize + j] = dataMsg.data[
                                j];
95                  }
96          }
97
98          //写入文件
99          cout << "\n文件传输成功, 开始写入文件" << endl;
100         FILE* outputfile;
101         outputfile = fopen(fileName, "wb");
102         if (fileBuffer != 0)
103         {
104                 fwrite(fileBuffer, fileSize, 1, outputfile);
105                 fclose(outputfile);
106         }
107         cout << "文件写入成功" << endl<<endl;
108         delete[] fileBuffer;//释放内存
109 }
```

下面展示一下 **recvMessage 函数**, 具体代码如下:

recvMessage 函数

```
1  //实现单个报文接收
2  bool recvMessage(Message& recvMsg, SOCKET serverSocket, SOCKADDR_IN
       clientAddr)
3  {
4          int AddrLen = sizeof(clientAddr);
```

```
5         while (1)
6         {
7                 int recvByte = recvfrom(serverSocket, (char*)&recvMsg, sizeof
                      (recvMsg), 0, (sockaddr*)&clientAddr, &AddrLen);
8                 if (recvByte > 0)
9                 {
10                        //成功收到消息，回复ACK报文
11                        if (recvMsg.check() && (recvMsg.SeqNum == initrelseq
                              + 1))
12                        {
13                                Message replyMessage;
14                                replyMessage.SrcPort = ServerPORT;
15                                replyMessage.DestPort = RouterPORT;
16                                replyMessage.flag += ACK;
17                                replyMessage.SeqNum=initrelseq++;
18                                replyMessage.AckNum = recvMsg.SeqNum;
19                                replyMessage.setCheck();
20                                sendto(serverSocket, (char*)&replyMessage,
                                      sizeof(replyMessage), 0, (sockaddr*)&
                                      clientAddr, sizeof(SOCKADDR_IN));
21                                cout << "server收到: seq␣=␣" << recvMsg.
                                      SeqNum << "的数据报文"<< endl;
22                                cout << "server发送: seq␣=␣"<<replyMessage.
                                      SeqNum<<", ack␣=␣" << replyMessage.AckNum
                                       << "的ACK报文，检验和"<<replyMessage.
                                      checkNum << endl;
23                                return true;
24                        }
25
26                        //如果seq值不正确，则丢弃报文，返回累计确认的ACK报文
                              (ack=initrelseq-1)
27                        else if (recvMsg.check() && (recvMsg.SeqNum !=
                              initrelseq + 1))
28                        {
29                                Message replyMessage;
30                                replyMessage.SrcPort = ServerPORT;
31                                replyMessage.DestPort = RouterPORT;
32                                replyMessage.flag += ACK;
33                                replyMessage.SeqNum = initrelseq;
34                                replyMessage.AckNum = initrelseq;
35                                replyMessage.setCheck();
36                                sendto(serverSocket, (char*)&replyMessage,
                                      sizeof(replyMessage), 0, (sockaddr*)&
                                      clientAddr, sizeof(SOCKADDR_IN));
37                                cout << "[累计确认（错误seq值）]␣server收到␣
                                      seq␣=␣" << recvMsg.SeqNum << "的数据报
                                      文，并发送␣ack␣=␣" << replyMessage.AckNum
                                       << "␣的累计确认的ACK报文" << endl;
```

```
                        }
                }
                else if (recvByte == 0)
                {
                        return false;
                }
        }
        return true;
}
```

4. 四次挥手核心代码

这里具体的协议设计，包括断开连接协议和超时重传协议已经在上文的协议设计中进行了具体的说明，下面讲一下实现：这里的实现流程与协议设计中说明的一样，在连接的时候必须是一端等待另一端的消息，也就是流量控制使用停等协议，收发报文利用的函数主要是 UDP 中常用的 **sendto 函数**和 **recvform 函数**。这部分代码与 3-1 实验中一致。

client 端 ：

client 四次挥手

```
//实现client的四次挥手
bool fourwayhandwave(SOCKET clientSocket, SOCKADDR_IN serverAddr)
{
        //省略其余与3-1实验相同代码
        return true;
}
```

server 端 ：

server 四次挥手

```
//实现server的四次挥手
bool fourwayhandwave(SOCKET serverSocket, SOCKADDR_IN clientAddr)
{
        //省略其余与3-1实验相同代码
        return true;
}
```

(三) 测试

1. makefile 编写

这里简单的编写了一个 makefile 来方便编译和运行，具体代码为了节省篇幅就不复制到报告中了，我已经放在了提交作业的压缩包中。这里 makefile 编写与 3-1 实验也一致。

2. 开启测试

首先按照 router 使用手册设置好参数，地址使用的是本地回环地址，router 端口号是 30000，服务器端口号是 40460，丢包率设置为 5%，延时设置为 10ms。

然后同时开启两个命令行，前后分别输入 **make server** 和 **make client** 命令，结果如下：

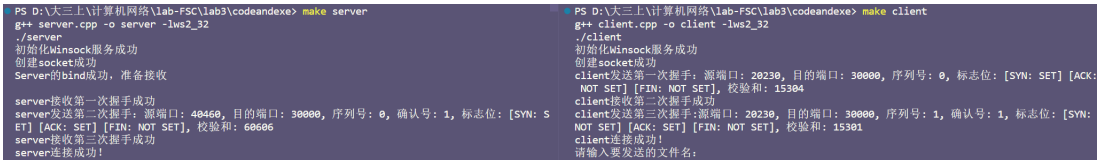

图 7: 三次握手建立连接

上图中展示了三次握手建立连接的过程，这部分输出与 3-1 实验一致，经过我的检查输出中的序列号和确认号等与协议设计中一致。然后这里我选择发送 1.jpg 文件，然后结果如下：

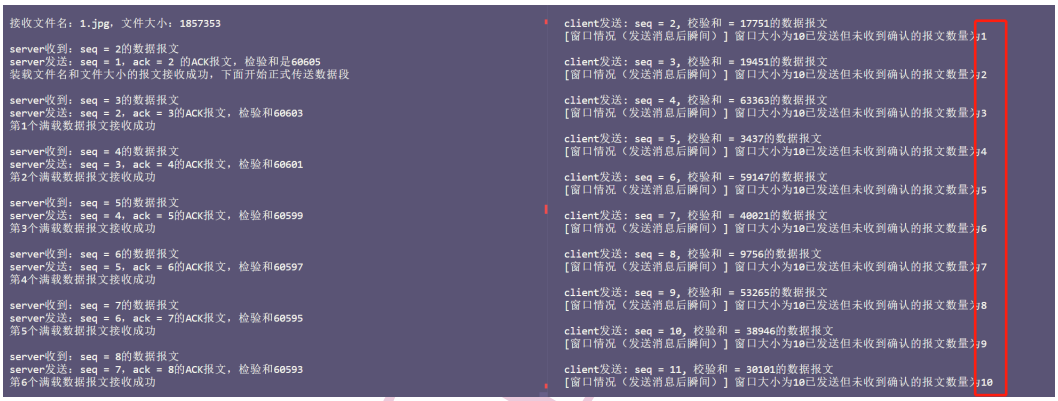

图 8: 数据传输输出（滑动窗口）

根据上图可以看到窗口（大小这里为 10）未满之前可以连续发送报文，窗口信息输出也正确。经过检查，剩余输出信息与协议设计中的一致，正确无误，接下来，我们来看一下如果发生了异常情况，我设计的机制是否可以正确执行，如图：

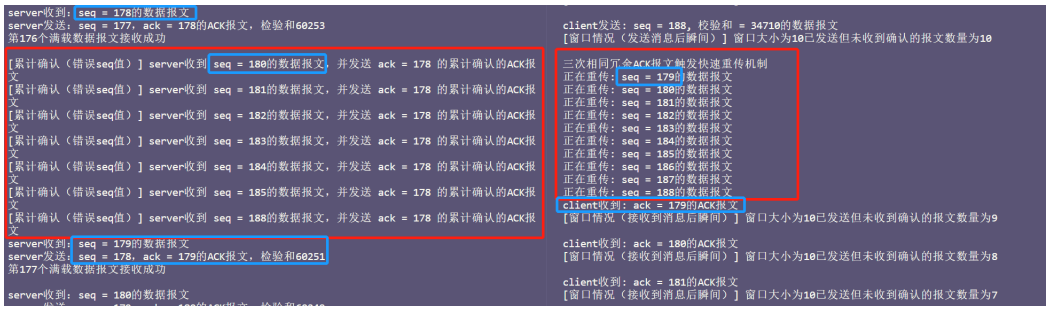

图 9: 快速重传

根据上图输出结果可以分析得出，这是发生了丢包或者失序的问题，根据上文的分析，丢包会触发快速重传，然后传输当前窗口内已发送未确认的所有数据报文，图中可以看到 server 在收到序列号为 178 的数据报文后直接收到序列号为 180 的数据报文，说明序列号为 179 的数据报文发生了失序或者丢包（从理论上分辨不出来到底哪种问题，但是这里大概率是丢包问题），可以

看到快速重传功能正确无误，而且重传后 server 也收到了序列号为 179 的数据报文，而且 client 也收到了对应的这个 ACK 报文并更新了窗口继续进行后续的发送过程。

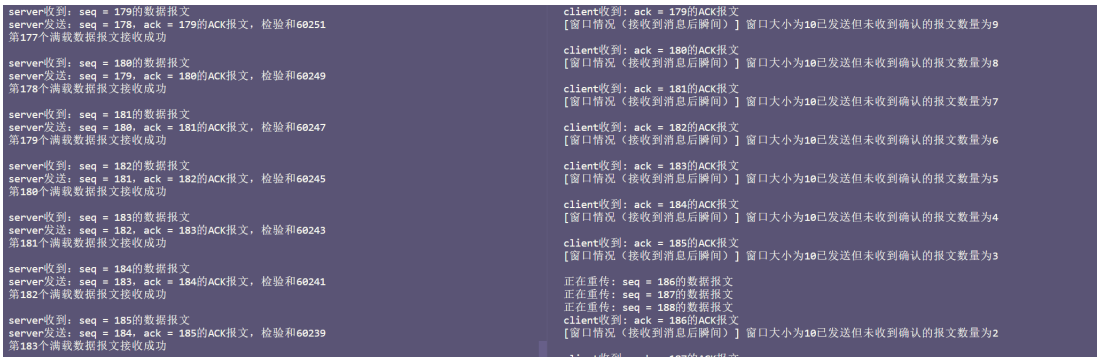

图 10: 累计确认

根据上图可以看出来，这里可以看到 client 端每收到一个正确的 ACK 报文就会更新一次窗口，这里可以看到每次确认到一个 ACK，窗口右移 1，代表这之前的数据报文都被确认接收。说明累计确认实现正确无误。

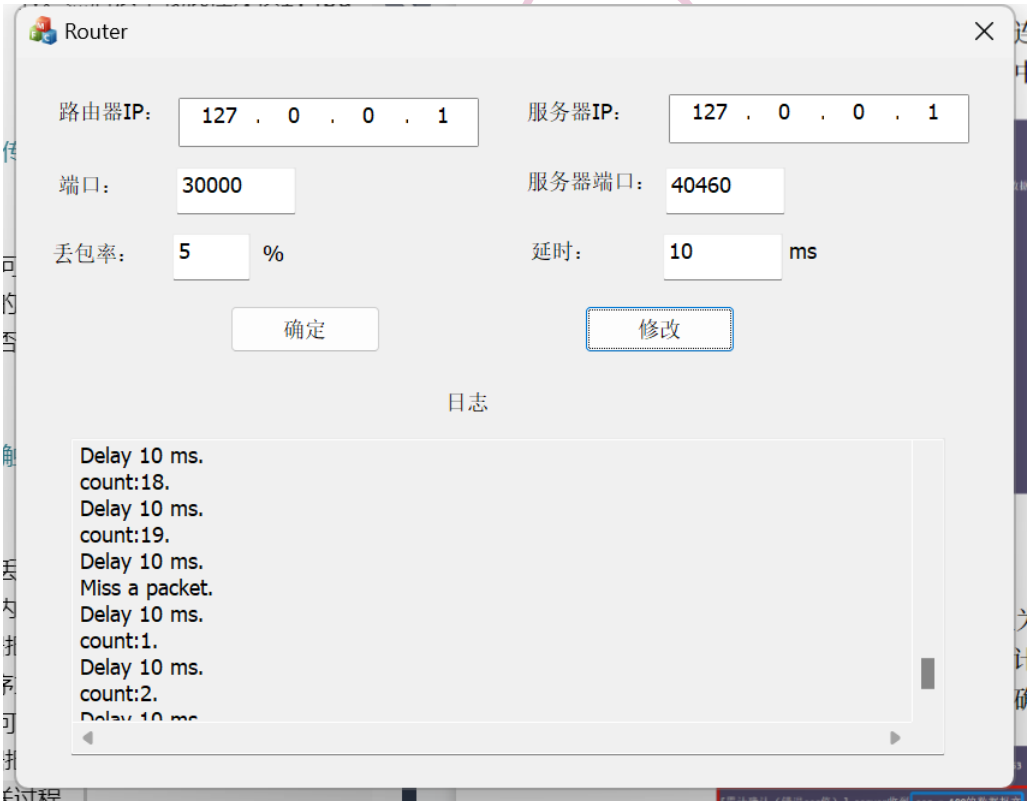

图 11: router 日志

这里设置丢包率为 5%，所以这里会每 count 到 19 就发生一次丢包，这里 router 日志也输出正确。延时输出也均正确。

接下来发送文件流程如上，然后到了关闭连接的流程，结果如下:

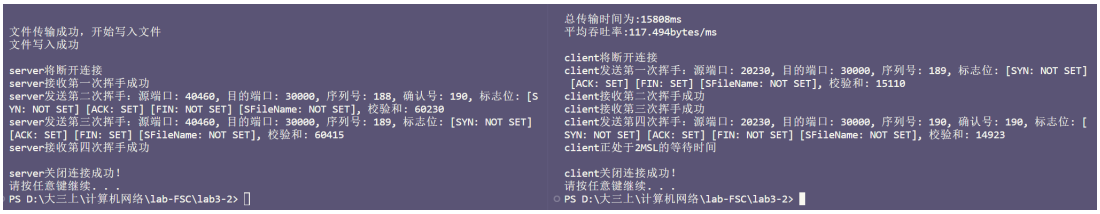

图 12: 四次挥手输出

这里并没有发生意外情况，实现了正确的断开连接的流程。剩余信息经过检查均与协议中的设计完全相符，正确无误。另外可以看到 client 端输出了总传输时间和平均吞吐率。

接下来我依次测试了 2.jpg、3.jpg 和 helloworld.txt 测试文件，均可以正确实现传输，同时这里的输出的文件名字和文件大小与原来完全一致，下面由于篇幅限制，仅展示 1.jpg 的对比图，如下：（蓝框表示传输后的文件，红框表示源文件）

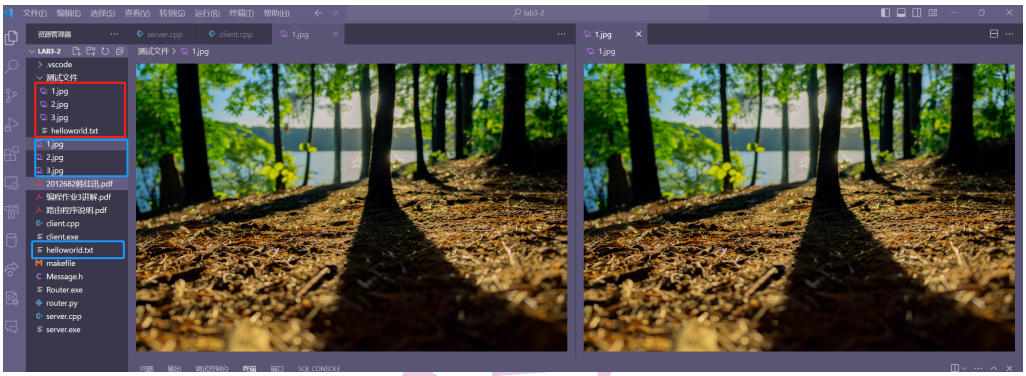

图 13: 1.jpg 图片传输前后对比

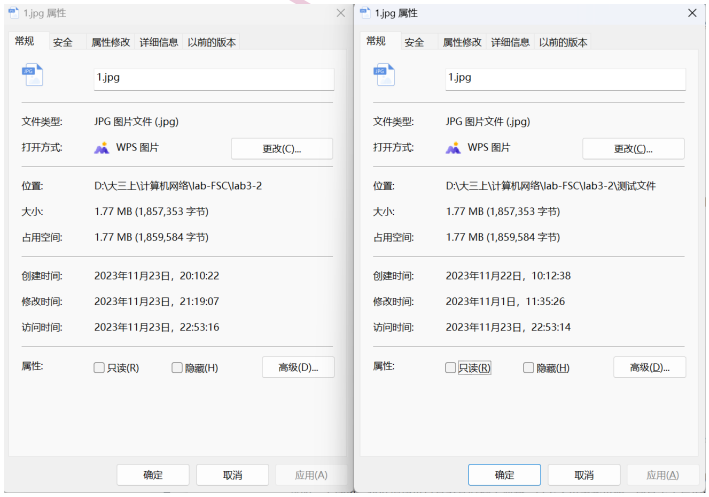

图 14: 1.jpg 图片传输前后属性对比

3. 性能测试分析（包含四个测试文件的指标表格）

首先展示一下在 5% 丢包率和 10ms 延时，以及我将超时重传时间设为 500ms 下 4 个测试文件对应的总传输时间和平均吞吐率，结果如下：

Time (ms)	Rate (bytes/ms)	Name
15808	117.494	1.jpg
52463	112.432	2.jpg
116468	102.766	3.jpg
13407	123.503	helloworld.txt

表 3: 测试文件的总传输时间和平均吞吐率

可以看到这里参数设置的一致的情况下，可以看到平均吞吐率是相近的，但是总传输时间会和文件大小成正相关。

我测试了针对同一个文件（这里是 1.jpg），这里我采用了对照实验的思想，即控制单一变量。同时由于网络和 router 的波动性，为了确保数据的稳定性，我下面展示的每个数据都是同样条件下**重复测试 5 次**取的平均数。(时延保留整数，吞吐率保留 6 位有效数字)

首先进行测试丢包率对速率的影响，这里设置延时为 0ms（防止其影响），其他参数均一致。将丢包率分别设置为 0、1、3、6、10% 进行测试，结果如下：

Time (ms)	Rate (bytes/ms)	丢包率
308	6030.37	0%
4486	414.033	1%
5942	312.580	3%
6210	299.091	6%
6691	277.598	10%

表 4: 滑动窗口机制下的 1.jpg 不同丢包率的测试对比（有快速重传）

可以看到在丢包率从 0% 到 1% 的时候，传输速率发生了一个突降，这是由于开始触发快速重传了，由于在本次实验中每个触发重传都会将在发送端缓冲区的已发送未确认的报文全部重发，所以实际上会多发送远多于原来报文数量的百分之一数量的报文，这大大降低了时间。然后发现之后随着丢包率的递增，传输速率缓慢下降，下降是由于重传的次数和报文数量增加了，所以这是合理的。至于为什么是缓慢，是因为虽然重传的频率增加了，但是在这个重传机制的实现下，每次都会重传所有已发送未确认的报文，所以由于丢包率增加而增加多发送报文的数量增加并不明显，造成了速率下降并不明显。

接下来进行测试时延对速率的影响，这里设置丢包率为 0%（防止其影响），其他参数均一致。将延时分别设置为 0、10、20、30、40、50ms 进行测试，结果如下：

Time (ms)	Rate (bytes/ms)	时延 (ms)
308	6030.37	0
11940	155.557	10
13968	132.972	20
17778	104.475	30
24019	77.3285	40

表 5: 滑动窗口机制下的 1.jpg 不同时延的测试对比（有快速重传）

可以看到对于也是在从 0ms 到 10ms 的延时增加后传输速率有一个突降，这是合理的，因为每个报文都会延时，所以整个会有一个较大的延时。然后之后会发现速率是逐渐递减的，下降

速度几乎相等。

然后我又来探索了滑动窗口的大小对于传输性能的影响，这里如果将丢包率和时延都设置为 0 的话，会发现差别较小，所以这里我设定为丢包率为 1% 时延也设为 1ms 进行测试，窗口大小我分别设为 1（相当于停等机制）、4、8、16、32 进行测试，结果如下：

Time (ms)	Rate (bytes/ms)	滑动窗口大小
13251	140.167	1
12830	144.766	4
12446	149.233	8
12243	151.707	16
12442	149.281	32

表 6: 滑动窗口机制下的 1.jpg 不同窗口大小的测试对比（有快速重传）

可以发现这里虽然随着窗口大小变大，传输速率的趋势是增加的，但是可以发现有反转的情况，而且增加不多，这里我具体的探究了原因，主要是由于你在增加窗口大小的同时，重传的报文数量也增加了，所以一方面加快了，一方面也减速了，所以导致加速不明显。另一个原因就是我们的本次实验的 server 端的窗口大小是 1，是按序接收，并没有像真实的网络情况一样，可以同时接收多条报文然后回复一个 ACK 报文。这一点也让加速没有那么明显。

更加细致地数据结果对比会在实验 3-4 完成。

另一方面，由于本次实验利用的 router 程序是一个 MFC 程序，会有一定的延时，所以其实测试的结果也会有一定的误差。

4. 额外测试

这里我要进行一些验证程序高健壮性的测试，这里我将丢包率设置为 25%，延时设置为 50ms，测试 1.jpg 的传输，结果如下：

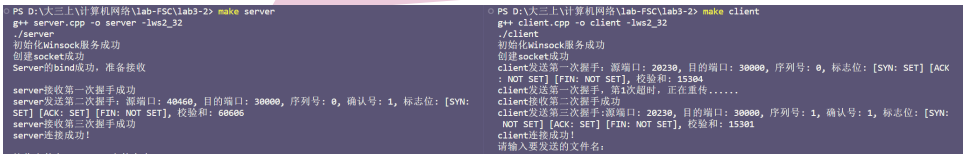

图 15: 三次握手

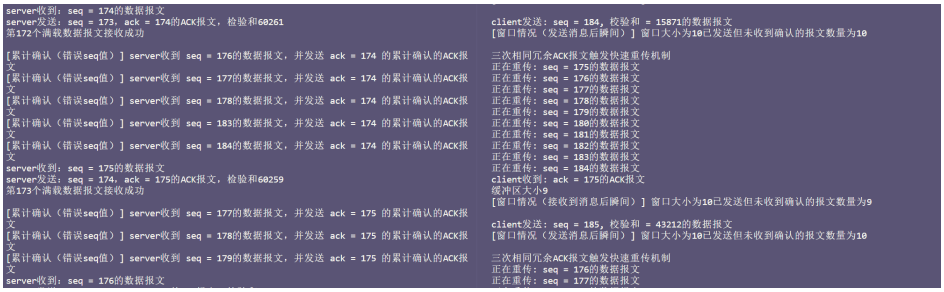

图 16: 数据传输

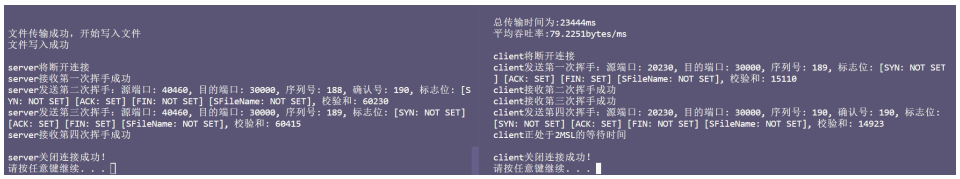

图 17: 四次挥手

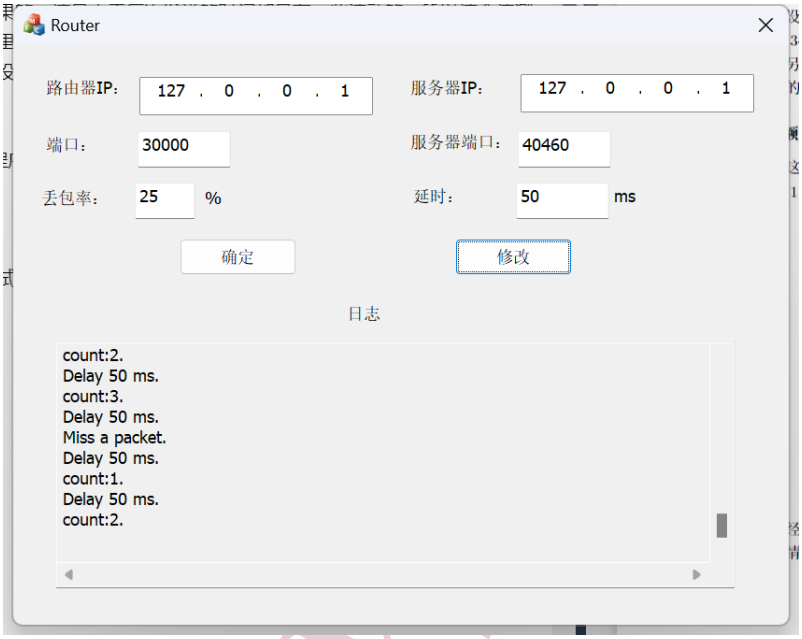

图 18: router 的输出

可以看到丢包率和延时提高后，传输速度明显下降。输出上文已经进行过仔细说明了，这里不进行赘述。经过检查发现，输出的消息是完全正确的。可以应对高丢包率和高延时的情况，而且也体现了一些意外情况的正确处理。体现了代码和协议设计的鲁棒性。

三、 总结与问题分析

这次实验在编码方面难度较高，主要遇到的困难有两点，第一点就是考虑到各种的意外情况并进行逻辑上的 debug，这一点耗费了我很长时间，问题最后都解决了。

第二点就是 client 端双线程的设计，这里需要注意对共享变量的更改和访问，而且这里在开始我遇到了输出被打乱的情况，这是由于多线程会在访问控制台的时候会产生冲突。之后我也进行了适当的加锁和解锁来维护，最后正确地实现了协议，这里加锁我是用的**互斥锁**。

还有一个小问题就是设置非阻塞的问题，需要在初始化 socket 的时候将其设置为非阻塞，由于 recvfrom 函数是阻塞的，设置非阻塞后就可以在 while 循环里等候消息的同时判断是否延时，而非一直被阻塞到接收消息的地方。

另一方面，我还测试了**非常高时延**的情况，例如将时延设置为 1000+，程序可能会出现问题，这是由于一些旧的或者重传的包在不正确的时候发送到对方，会导致状态机的混乱。但是实际上这也是正常现象，数据包传输受到多个变量的影响。因素如路由器的处理能力、网络的拥挤程度和物理距离都会影响数据包的传输时间。由于这些因素都有不确定性，网络中的延迟也因此变得难以预测且频繁变动。这种不确定性可能导致数据包在到达目标时出现延迟，进而可能对接收端

的缓冲区造成损害，引发一些意外状况。所以现在我们也在计算机网络领域做进一步的设计，增强鲁棒性。

在这个实验过程，我对 socket 编程更加的熟悉，使用起来也更加的得心应手。同时对于协议设计有了更加深刻的认识和了解。

整个过程锻炼了我的分析能力也磨练了我的耐心，可以说是非常不错的一次针对 GBN 机制的实现练习实验。