



南开大学
Nankai University

基于 UDP 服务可靠传输协议 编程实现 (3-3)

姓 名： 边 笛

学 号： 2012668

学 院： 计算机学院

目 录

一、 实验内容说明	1
二、 实验设计	1
1. 数据报套接字	1
2. 建立连接	2
3. 差错检验	3
4. 滑动窗口 GBN	4
5. 累计确认	5
6. rdt3.0 超时重传	6
7. 拥塞控制——reno 算法	7
8. 丢包设置	8
9. 数据报传输	9
10. 日志输出	9
三、 程序流程说明	10
四、 运行结果展示	11
五、 问题与分析	13

一、实验内容说明

本实验要求利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、确认重传等。流量控制采用滑动窗 GBN+reno 算法拥塞控制，完成给定测试文件的传输。

二、实验设计

1. 数据报套接字

实验使用用户数据报协议 UDP 作为传输协议，它是一种无连接、不可靠的传输协议。为实现传输协议，在初始化套接字是将协议设置为 `IPPROTO_UDP`，数据传输使用 `sendto`、`recvfrom` 实现无连接数据传输。

UDP 是面向报文的，在 3-1 中对于传输的数据报做了如下设计：

Scr_Port 源端口号	Dst_Port 目的端口号	
seq 序列号	ack 确认号	
datalen 数据长度	Flags 标志位	Checksum 校验和
Data 数据		

在实验 3-2 中将停等机制改为了基于滑动窗口的流量控制机制，对此在数据传输时需要对于窗口大小进行传输。对此修改了数据报结构体，增加了表示滑动窗口大小的窗口位：

Scr_Port 源端口号	Dst_Port 目的端口号	
seq 序列号	ack 确认号	
datalen 数据长度	Flags 标志位	Checksum 校验和
Winsize 窗口大小	Data 数据	

同时服务器与客户端产生用于进行数据校验的伪首部，与数据报一起用于校验和的计算，对于数据的正确性作检查。

Scr_IP 源地址		
Dst_IP 目的地址		
Zero 零位	Protocol 协议	Len 长度

```

struct Message
{
    unsigned short src_port;
    unsigned short dst_port;
    unsigned int seq;
    unsigned int ack;
    unsigned int datalen = 0;
    unsigned int Winsize = Max_window;
    unsigned short Flags = 0;
    unsigned short CheckSum;
    char Data[8192]{};
};

struct Header
{
    unsigned long src_IP;
    unsigned long dst_IP;
    char zero = 0;
    int Protocol = 17;
    int length = sizeof(struct Message);
};
    
```

对于序列号、确认号、标志位作说明：

序列号是发送信息的序号，每发送一条信息序列号+1；

确认号是对上一条接收到的消息的确认，确认号为上一条收到的消息的序列号；

标志位包含了具体的信息类型，目前设置了 6 位的含义（实际只用到了 5 位）：

从低到高依次为 FIN SYN RST PSH ACK isNAME：FIN 用于关闭连接；SYN 用于建立连接；RST 没用到；PSH 用于表示这是最后一条文件信息，可以进行输出了；isNAME 用于标记本条信息为文件名而非文件内容，不需要输出。

```

void setFin()
{
    Flags = Flags | 0x0001;
};
bool getFin()
{
    return Flags & 0x0001;
};

void setSyn()
{
    Flags = Flags | 0x0002;
};
bool getSyn()
{
    return Flags & 0x0002;
};

void setRst()
{
    Flags = Flags | 0x0004;
};
bool getRst()
{
    return Flags & 0x0004;
};

void setPsh()
{
    Flags = (Flags | 0x0008);
};
bool getPsh()
{
    return (Flags & 0x0008);
};

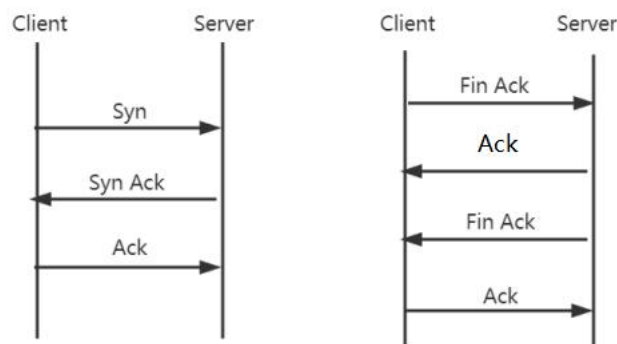
void setAck()
{
    Flags = Flags | 0x0010;
};
bool getAck()
{
    return Flags & 0x0010;
};

void setName()
{
    Flags = Flags | 0x0020;
};
bool getName()
{
    return Flags & 0x0020;
};
    
```

2. 建立连接

建立连接与断开连接过程参考了 TCP 的三次握手四次挥手进行设计。

如果想要建立连接或断开连接，需要完成如下消息传递：



建立连接是由客户端给服务器发送[SYN]表示请求连接；客户端做出[SYN , ACK]应答表示确认连

接请求，服务器也希望建立连接；客户端做[ACK]应答表示确认收到，连接由此建立。

Server:

```
[2022/12/02 22:26:26] [ Log ] Receive First Handshake
[2022/12/02 22:26:26] Client -> Server[ SYN ] Seq=0 Len=0 Win=0
[2022/12/02 22:26:26] [ Log ] Send Second Handshake
[2022/12/02 22:26:26] Server -> Client[ SYN ACK ] Seq=1 Ack=0 Len=0 Win=10
[2022/12/02 22:26:26] [ Log ] Receive Third Handshake
[2022/12/02 22:26:26] Client -> Server[ ACK ] Seq=1 Ack=1 Len=0 Win=0
```

Client:

```
[2022/12/02 22:26:26] [ Log ] Send First Handshake
[2022/12/02 22:26:26] Client -> Server[ SYN ] Seq=0 Len=0 Win=0
[2022/12/02 22:26:26] [ Log ] Receive Second Handshake
[2022/12/02 22:26:26] Server -> Client[ SYN ACK ] Seq=1 Ack=0 Len=0 Win=10
[2022/12/02 22:26:26] [ Log ] Send Third Handshake
[2022/12/02 22:26:26] Client -> Server[ ACK ] Seq=1 Ack=1 Len=0 Win=0
```

断开连接时客户端首先给服务器发送[FIN, ACK]，由 FIN 表示请求释放连接；服务器应答一个[ACK]确认收到；服务器再发送一个[FIN, ACK]表示释放连接；客户端最后回一个[ACK]做确认，连接关闭。

Server:

```
[2022/12/02 22:27:35] Receive First Handwave
[2022/12/02 22:27:35] Client -> Server[ FIN ACK ] Seq=3 Ack=2 Len=0 Win=0
[2022/12/02 22:27:35] [ Log ] Send Second Handwave
[2022/12/02 22:27:35] Server -> Client[ ACK ] Seq=3 Ack=3 Len=0 Win=10
[2022/12/02 22:27:35] [ Log ] Send Third Handwave
[2022/12/02 22:27:35] Server -> Client[ FIN ACK ] Seq=4 Ack=3 Len=0 Win=10
[2022/12/02 22:27:35] [ Log ] Receive Fourth Handwave
[2022/12/02 22:27:35] Client -> Server[ ACK ] Seq=4 Ack=4 Len=0 Win=0
```

Client:

```
[2022/12/02 22:27:35] [ Log ] Send First Handwave
[2022/12/02 22:27:35] Client -> Server[ FIN ACK ] Seq=3 Ack=2 Len=0 Win=0
[2022/12/02 22:27:35] [ Log ] Receive Second Handwave
[2022/12/02 22:27:35] Server -> Client[ ACK ] Seq=3 Ack=3 Len=0 Win=10
[2022/12/02 22:27:35] Receive Third Handwave
[2022/12/02 22:27:35] Server -> Client[ FIN ACK ] Seq=4 Ack=3 Len=0 Win=10
[2022/12/02 22:27:35] Send Fourth Handwave
[2022/12/02 22:27:35] Client -> Server[ ACK ] Seq=4 Ack=4 Len=0 Win=0
```

3. 差错检验

为保证数据的可靠性，需要进行差错检验。在这里使用纠错码进行差错检验。使用到前面说明的数据报与伪首部来计算校验和。

在数据传输之前，需要借助 Message 结构体的 void setChecksum(struct Header* h) 进行校验和的设置，基本思路如下：

1. 产生伪首部，将设置校验和域清零。
2. 将伪首部与数据包一起看成 16 位整数序列，进行 16 位二进制反码求和。
3. 将其算结果取反写入校验和域段。

```

void Message::setChecksum(struct Header* h)
{
    this->CheckSum = 0;

    unsigned long sum = 0;
    int i=0;
    int count = sizeof(struct Header) / 2;
    while(count-->0)
    {
        sum += ((unsigned short*)h)[i];
        i++;
        if (sum & 0xffff0000)
        {
            sum &= 0x0000ffff;
            sum++;
        }
    }

    i=0;
    count = sizeof(struct Message) / 2;
    while(count-->0)
    {
        sum += ((unsigned short*)this)[i];
        i++;
        if (sum & 0xffff0000)
        {
            sum &= 0x0000ffff;
            sum++;
        }
    }
    this->CheckSum = ~(sum & 0xffff);
}
    
```

在接收到数据报后，接收端需要对数据报进行检查，对整个数据报反码求和。根据校验和的设置，可以判断当相加计算结果位全部为 1 说明没有检测到错误。基本思路如下：

1. 产生伪首部。
2. 按 16 位整数序列反码求和。
3. 判断计算结果，如果全 1 说明没有检测到差错，返回 true。

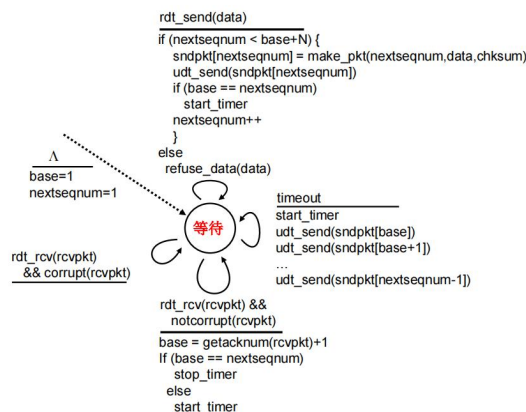
```

bool Message::Check(struct Header* h)
{
    unsigned long sum = 0;
    int i=0;
    int count = sizeof(struct Header) / 2;
    while(count-->0)
    {
        sum += ((unsigned short*)h)[i];
        i++;
        if (sum & 0xffff0000)
        {
            sum &= 0x0000ffff;
            sum++;
        }
    }

    i=0;
    count = sizeof(struct Message) / 2;
    while(count-->0)
    {
        sum += ((unsigned short*)this)[i];
        i++;
        if (sum & 0xffff0000)
        {
            sum &= 0x0000ffff;
            sum++;
        }
    }
    return sum == 0x0000ffff;
}
    
```

4. 滑动窗口 GBN

在进行数据传输时使用滑动窗口 GBN 的方法进行传输。GBN 状态机如下：



发送端的发送不依赖于收到确认，而是采用连续发送的方式，只要滑动窗口没有满，就可以

发送，如果窗口已满，就等待收到 ACK 窗口滑动后再次发送。因此发送端接受和发送的完成需要使用多线程完成，一条线程用于数据发送，另一条线程用于确认信息的接收。

```
struct Message sendcontent{Client_Port , Server_Port , lastsendSEQ+1 , lastrecvSEQ };
sendcontent.dataLen = contentlen;
sendcontent.setAck();
sendcontent.setData(content);
sendcontent.Winsize = cwnd - (lastrecvSEQ - lastrecvACK) - 1;
sendcontent.setChecksum(&sendHeader);
bool LOSS = LossPackage();
int s;
if(!LOSS)
{
    s = sendto(Client, (char*) &sendcontent, sizeof(struct Message), 0, (sockaddr*)&Server_addr, 1 );
}
else
{
    printTime();
    cout<<"Loss Seq"<<lastsendSEQ+1<<endl;;
}
if(!LOSS) printMess(sendcontent,SEND);
sendQ.push_back(sendcontent);
lastsendSEQ = sendcontent.seq; 窗口上界
```

图 1 发送端的发送线程

```
void Recvng()
{
    while(1)
    {
        struct Message recvcontent;
        //接收
        int r = recvfrom(Client, (char*) &recvcontent, sizeof(struct Message), 0, (sockaddr*)&Server_addr, &l);

        if( r!= SOCKET_ERROR && recvcontent.Check(&recvHeader) && recvcontent.Flags == 16 && recvcontent.seq >= lastrecvSEQ + 1 && recvcontent.ack <= lastsendSEQ && recvcontent.ack > lastrecvACK)
        {
            printMess(recvcontent,ISEND);
            lastrecvSEQ = recvcontent.seq;
            lastrecvACK = recvcontent.ack; 窗口下界
            recvWin = recvcontent.Winsize;
            //计时
            filetimer = clock();
            //正确的ACK
            switch(STATE)...
```

图 2 发送端的接收线程

由于队列 queue 的遍历需要弹出队列内容再放入，在多次线程程序中会涉及到两条线程同时对队列进行操作的情况，加锁之后 debug 仍旧有些困难，于是本次实验对于 3-2 的代码进行了修改，滑动窗口改为使用动态数组 vector 维护，在发送消息后会将消息加入到动态数组中，更新 lastsendSEQ。每接收到一条消息就更新 lastrecvACK，实现累计确认。滑动窗口中的消息序列号属于 [lastrecvACK,lastsendSEQ]。每接收或者发送一条消息对于 lastsendSEQ、lastrecvACK 的更新就相当于滑动窗口的移动。如果超时则需要重传所有已发送待确认的数据，也就是 vector 中序列号属于 [lastrecvACK,lastsendSEQ] 的消息。

5. 累计确认

接收端确认采用累计确认方式。发送端在收到确认信息后会滑动窗口，将序列号小于等于确认号的信息移出窗口，即 $ack = x+1$ 代表对于序列号为 $x+1$ 之前的所有信息在接收方都已经确认接收完毕，接收方期望发送方发送 $x+1$ 及其后面的消息。累计确认部分在图 2（上图）中标出。

6. rdt3.0 超时重传

确认重传采用 rdt3.0 的超时重传方式。

在程序中设置了一个最大等待时间，如果在接收到一条消息后的最大等待时间内没有收到下一条消息，就认为是超时。因此在收到消息后会开始计时，如果超过最大等待时间没有收到下一条正确的消息就会进行重传，重传时重传所有未得到确认的已发送消息，也就是 `vector` 中序列号属于 `[lastrecvACK, lastsendSEQ]` 的消息。

在发送端的接收线程中会对确认信息做判断，只有接收到校验和、序列号、标志位、确认号都正确合理的确认信息后才会对滑动窗口进行移动，其余情况都持续计时，并判断是否超时。超时重传涉及到本次实验完善的 `reno` 算法，具体的操作会在 `Reno` 算法部分进行说明，这里只做代码展示。

```
if(clock()-filetime >= Max_waitTime)
{
    //超时
    switch(STATE)
    {
        case SLOWSTART:
        {
            ssthresh = cwnd/2;
            dupACKcount = 0;
            cwnd = 1*MSS;
            Resend();
            if(cwnd >= ssthresh)
            {
                STATE = AVOID;
            }
            break;
        }
        case AVOID:
        {
            ssthresh = cwnd / 2;
            dupACKcount = 0;
            cwnd = 1*MSS;
            STATE = SLOWSTART;
            Resend();
            break;
        }
        case FASTRECO:
        {
            ssthresh = cwnd / 2;
            dupACKcount = 0;
            cwnd = 1 *MSS;
            STATE = SLOWSTART;
            Resend();
            break;
        }
    }
}
```

图 3 发送端重传判断

```
void Resend()
{
    std::lock_guard<std::mutex> lockGuard(bufferMutex);
    if(lastrecvACK == lastsendSEQ) return;
    //遍历队列重传
    for(int i=lastrecvACK - Packstart + 1; i <= lastsendSEQ - Packstart; i++)
    {
        struct Message resend = sendQ[i];
        int s = sendto(Client, (char*) &resend, sizeof(struct Message), 0, (sockaddr*)&Server_addr, 1);
        printMess(resend,SEND);
    }
}
```

图 4 发送端重传函数

同样在接收端只接收校验和、序列号、标志位、确认号都正确合理的数据信息，如果发送端发送数据时出现了丢包问题，接收端会由于没有接收到目标序列号的数据而不进行数据接收，因此也并不会发送确认信息，最终发送端会超时重传。

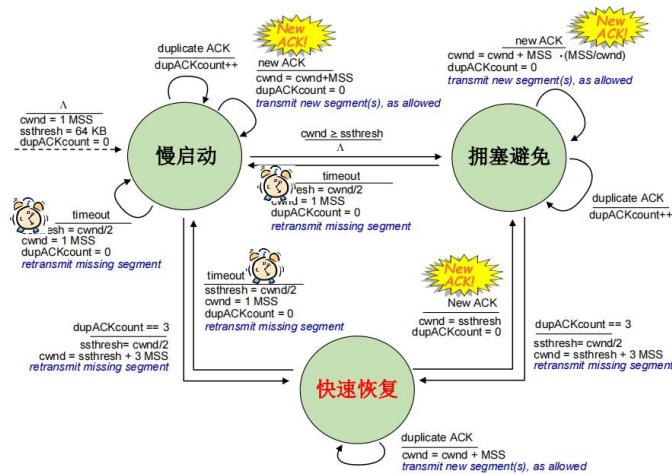
如果在发送端超过等待时间仍未收到发送端发来的数据，接收端也会重传确认信息。在累计确认的情况下，发送端只需要发送最后一次发送的 ACK 消息。

```
if(clock()-filetimer > Max_waitTime)
{
    int s = sendto(Server, (char*)&sendQ.back(), sizeof(struct Message), 0, (sockaddr*)&Client_addr, 1);
    std::cout<<"[ Log ] Resend File Data Block Ack"<<endl;
    printMess(sendQ.back(),SEND);
    continue;
}
```

图 5 接收端重传

7. 拥塞控制——reno 算法

reno 算法的状态图如下：



reno 算法是一种基于窗口的拥塞控制算法，通过拥塞窗口的增大或减小控制发送速率。在程序中预先设定了 MSS（Maximum Segment Size）和上限 ssthresh，在接受消息的过程中记录并根据算法维护连续接收到的冗余 ACK 报文数 dupACKcount。

初始处于慢启动阶段， $cwnd = 1MSS$ ，每收到一个正确的 ACK 报文， $cwnd$ 增加 $1MSS$ ， $dupACKcount$ 归 0；如果慢启动过程中接收到冗余 ACK， $dupACKcount$ 增加 1；如果连续接收到 3 个冗余 ACK，也就是 $dupACKcount=3$ ，会进入快速恢复阶段， $ssthresh$ 设为 $cwnd/2$ ， $cwnd$ 设为 $ssthresh+3MSS$ ，重传待确认的已发送消息；若在出现超时，会将 $ssthresh$ 设为 $cwnd/2$ ， $cwnd$ 重新设为 $1MSS$ ， $dupACKcount$ 归 0，进行超时重传；若随着 $cwnd$ 与 $ssthresh$ 的变化， $cwnd \geq ssthresh$ 了，会进入拥塞避免阶段。

在拥塞避免阶段，如果收到正确的 ACK 报文， $cwnd$ 增加 $MSS * (MSS/cwnd)$ ， $dupACKcount$ 归 0；若接收到冗余 ACK， $dupACKcount$ 增加 1；如果连续接收到 3 个冗余 ACK，也就是 $dupACKcount=3$ ，会进入快速恢复阶段， $ssthresh$ 设为 $cwnd/2$ ， $cwnd$ 设为 $ssthresh+3MSS$ ，重传待确认的已发送消息；若出现超时，会将 $ssthresh$ 设为 $cwnd/2$ ， $cwnd$ 重新设为 $1MSS$ ， $dupACKcount$ 归 0，进行超时重传，并转移到慢启动阶段。

在快速恢复阶段，如果接收到正确的 ACK 报文， $cwnd$ 设为 $ssthresh$ ， $dupACKcount$ 归 0，进入

拥塞避免阶段：若接收到冗余 ACK，cwnd 增加 MSS；若出现超时，会将 ssthresh 设为 cwnd/2cwnd 设为 ssthresh+3MSS，dupACKcount 归 0，进行超时重传，并转移到慢启动阶段。

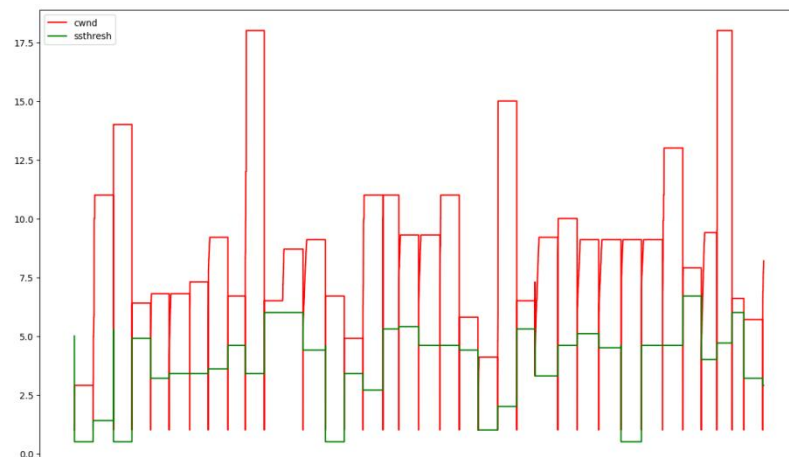
对应的代码实现如下：

```
//正确的ACK
switch(STATE)
{
    case SLOWSTART:
    {
        dupACKcount = 0;
        cwnd = cwnd + MSS;
        if(cwnd >= ssthresh)
        {
            STATE = AVOID;
        }
        break;
    }
    case AVOID:
    {
        dupACKcount = 0;
        cwnd = cwnd + MSS*MSS/cwnd;
        break;
    }
    case FASTRECO:
    {
        cwnd = ssthresh;
        dupACKcount = 0;
        STATE = AVOID;
        break;
    }
}

//dupACK
switch(STATE)
{
    case SLOWSTART:
    {
        dupACKcount ++;
        if(dupACKcount == 3)
        {
            ssthresh = cwnd / 2;
            cwnd = ssthresh + 3*MSS;
            STATE = FASTRECO;
            Resend();
        }
        break;
    }
    case AVOID:
    {
        dupACKcount ++;
        if(dupACKcount == 3)
        {
            ssthresh = cwnd / 2;
            cwnd = ssthresh + 3*MSS;
            STATE = FASTRECO;
            Resend();
        }
        break;
    }
    case FASTRECO:
    {
        cwnd = cwnd + MSS;
        break;
    }
}

//超时
switch(STATE)
{
    case SLOWSTART:
    {
        ssthresh = cwnd / 2;
        dupACKcount = 0;
        cwnd = 1*MSS;
        Resend();
        if(cwnd >= ssthresh)
        {
            STATE = AVOID;
        }
        break;
    }
    case AVOID:
    {
        ssthresh = cwnd / 2;
        dupACKcount = 0;
        cwnd = 1*MSS;
        STATE = SLOWSTART;
        Resend();
        break;
    }
    case FASTRECO:
    {
        ssthresh = cwnd / 2;
        dupACKcount = 0;
        cwnd = 1 *MSS;
        STATE = SLOWSTART;
        Resend();
        break;
    }
}
```

在程序中记录 ssthresh 和 cwnd 的变化情况，以传输 3.jpg 为例，获取数据绘制为下图：



可以看到 cwnd 和 ssthresh 的变化情况，通过该图也能大致判断各个时刻流量的变化情况。

8. 丢包设置

在头文件中编写了丢包函数：

```
bool LossPackage()
{
    return rand() % 100 < loss_rate;
}
```

在发送数据时随机以设置的丢包率 loss_rate 丢包

```
bool LOSS = LossPackage();
int s;
if(!LOSS)
{
    s = sendto(Client, (char*) &sendcontent, sizeof(struct Message), 0, (sockaddr*)&Server_addr, 1);
}
else
{
    printTime();
    cout<<"Loss Seq"<<lastsendSEQ+1<<endl;
}
```

运行程序进行检验。

2022/12/02 22:41:42]	Server -> Client	ACK	Seq=177	Ack=177	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=185	Ack=177	Len=8192	Win=6
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=178	Ack=178	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=186	Ack=178	Len=8192	Win=6
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=179	Ack=179	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=187	Ack=179	Len=8192	Win=6
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=180	Ack=180	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=188	Ack=180	Len=8192	Win=7
2022/12/02 22:41:42]	Loss Seq189	丢包				
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=181	Ack=181	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=190	Ack=181	Len=8192	Win=7
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=182	Ack=182	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=191	Ack=182	Len=8192	Win=7
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=183	Ack=183	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=192	Ack=183	Len=8192	Win=7
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=184	Ack=184	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=193	Ack=184	Len=8192	Win=7
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=185	Ack=185	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=194	Ack=185	Len=8192	Win=7
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=186	Ack=186	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=195	Ack=186	Len=8192	Win=7
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=187	Ack=187	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=196	Ack=187	Len=8192	Win=7
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=188	Ack=188	Len=0	Win=10
2022/12/02 22:41:42]	Client -> Server	ACK	Seq=197	Ack=188	Len=8192	Win=8
2022/12/02 22:41:42]	Server -> Client	ACK	Seq=189	Ack=188	重传	Len=8192 Win=8
2022/12/02 22:41:43]	Client -> Server	ACK	Seq=189	Ack=180	Len=8192	Win=7
2022/12/02 22:41:43]	Client -> Server	ACK	Seq=190	Ack=181	Len=8192	Win=7
2022/12/02 22:41:43]	Client -> Server	ACK	Seq=191	Ack=182	Len=8192	Win=7

如图，再丢包后进行了重传。

9. 数据报传输

在进行传输时，数据都存储在 **Message** 的 **Data** 里。但由于文件大小不定，数据很可能无法由一个数据报完成传输。对此需要对文件进行划分传输。

首先计算所需数据报的个数：使用文件总长 除以 **Data** 的最大长度 向上取整。

由于可能不能整除，最后一个数据报的数据段就有可能是不满的，需要补 0 补全。

此外，最后一个数据报表示文件传输结束，需要单独设置 **Psh** 标志位，用以告诉接收端数据传输完毕可以进行文件数据的读取与输出了。

对此在发送端需要对最后一个数据报进行单独置位；在接收端接收数据报时要注意判断 **Psh** 标志位，即使对完成传输的文件进行读取。

由于本实验完成的是文件传输，在发送文件内容之前，首先发送了一条 **Data** 存储着传输文件文件名的数据报作为数据头，标示了传输文件的信息。

10. 日志输出

对于文件传输的关键信息比如完成第几次握手、挥手，文件读入、文件传输开始或者结束都

有对应的日志输出，传递信息时会输出信息信息，包括该信息由谁发送给谁、标志位是什么、序列号确认号都是多少、data 段的长度、窗口大小等。

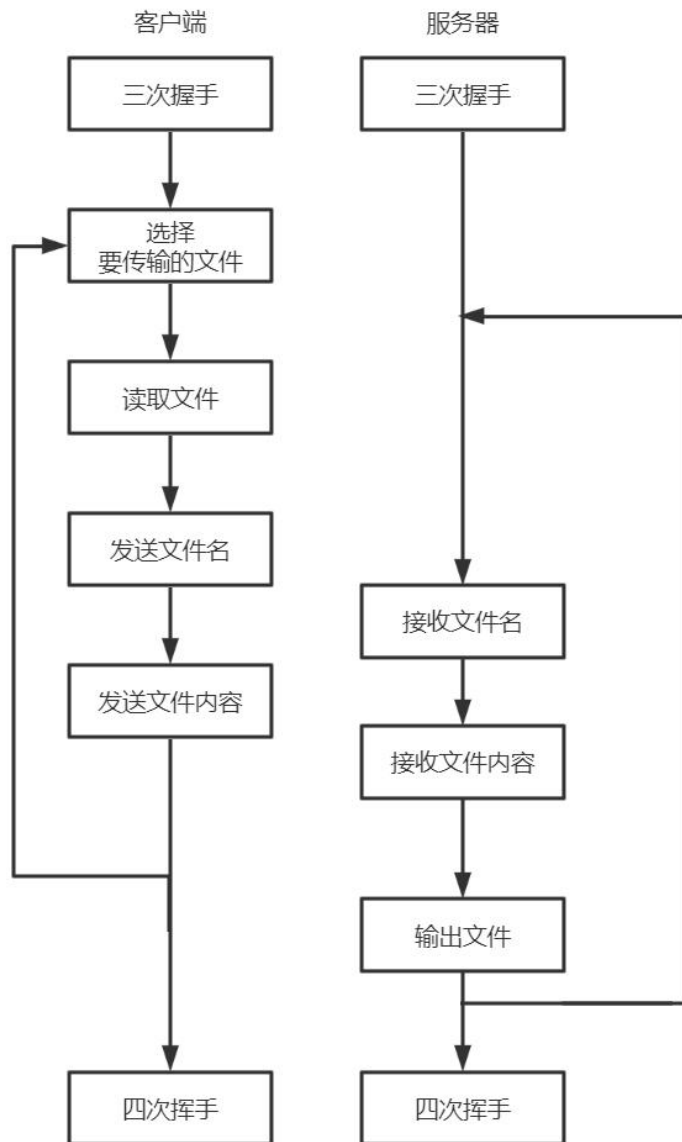
此外后续为了对 reno 算法做验证，还在程序中实时输出了 cwnd 大小。

```
[2022/12/11 12:14:17] cwnd: 10
[2022/12/11 12:14:17] Server -> Client[ ACK ] Seq=185 Ack=185 Len=0 Win=10
[2022/12/11 12:14:17] cwnd: 10
[2022/12/11 12:14:17] Server -> Client[ ACK ] Seq=186 Ack=186 Len=0 Win=10
[2022/12/11 12:14:17] cwnd: 10
[2022/12/11 12:14:17] Server -> Client[ ACK ] Seq=187 Ack=187 Len=0 Win=10
[2022/12/11 12:14:17] cwnd: 10
[2022/12/11 12:14:17] Server -> Client[ ACK ] Seq=188 Ack=188 Len=0 Win=10
```

在文件传输过程中还进行了计时，输出了每个文件的发送用时。

三、程序流程说明

客户端与服务器的运行流程如下所示：



四、运行结果展示

对于测试样例中的四个文件都进行了传输：

三次握手

Server:

```
[2022/12/02 22:26:26] [ Log ] Receive First Handshake
[2022/12/02 22:26:26] Client -> Server[ SYN ] Seq=0   Len=0   Win=0
[2022/12/02 22:26:26] [ Log ] Send Second Handshake
[2022/12/02 22:26:26] Server -> Client[ SYN ACK ] Seq=1 Ack=0   Len=0   Win=10
[2022/12/02 22:26:26] [ Log ] Receive Third Handshake
[2022/12/02 22:26:26] Client -> Server[ ACK ] Seq=1 Ack=1   Len=0   Win=0
```

Client:

```
[2022/12/02 22:26:26] [ Log ] Send First Handshake
[2022/12/02 22:26:26] Client -> Server[ SYN ] Seq=0   Len=0   Win=0
[2022/12/02 22:26:26] [ Log ] Receive Second Handshake
[2022/12/02 22:26:26] Server -> Client[ SYN ACK ] Seq=1 Ack=0   Len=0   Win=10
[2022/12/02 22:26:26] [ Log ] Send Third Handshake
[2022/12/02 22:26:26] Client -> Server[ ACK ] Seq=1 Ack=1   Len=0   Win=0
```

传输 1.jpg

Server

```
[2022/12/02 22:45:34] Server -> Client[ ACK ] Seq=228 Ack=228   Len=0   Win=10
[2022/12/02 22:45:34] Client -> Server[ PSH ACK ] Seq=229 Ack=222   Len=5961 Win=5
[2022/12/02 22:45:34] Server -> Client[ ACK ] Seq=229 Ack=229   Len=0   Win=10
[2022/12/02 22:45:34] [ Log ] File 1.jpg has been written to destination folder.
```

Client

```
[2022/12/02 22:45:34] Client -> Server[ ACK ] Seq=228 Ack=221   Len=8192 Win=5
[2022/12/02 22:45:34] Client -> Server[ PSH ACK ] Seq=229 Ack=222   Len=5961 Win=5
[2022/12/02 22:45:34] Server -> Client[ ACK ] Seq=228 Ack=228   Len=0   Win=10
[2022/12/02 22:45:34] Server -> Client[ ACK ] Seq=229 Ack=229   Len=0   Win=10
[2022/12/02 22:45:34] [ Log ] 1.jpg transmission completed. Cost 6.92s
```

传输 2.jpg

Server

```
[2022/12/02 22:47:18] Server -> Client[ ACK ] Seq=722 Ack=722   Len=0   Win=10
[2022/12/02 22:47:18] Client -> Server[ PSH ACK ] Seq=723 Ack=715   Len=265   Win=7
[2022/12/02 22:47:18] Server -> Client[ ACK ] Seq=723 Ack=723   Len=0   Win=10
[2022/12/02 22:47:18] [ Log ] File 2.jpg has been written to destination folder.
```

Client

```
[ ACK ] Seq=716 Ack=716   Len=0   Win=10
[2022/12/02 22:47:18] Client -> Server[ PSH ACK ] Seq=723 Ack=715   Len=265   Win=7
[2022/12/02 22:47:18] Server -> Client[ ACK ] Seq=717 Ack=717   Len=0   Win=10
[2022/12/02 22:47:18] Server -> Client[ ACK ] Seq=718 Ack=718   Len=0   Win=10
[2022/12/02 22:47:18] Server -> Client[ ACK ] Seq=719 Ack=719   Len=0   Win=10
[2022/12/02 22:47:18] Server -> Client[ ACK ] Seq=720 Ack=720   Len=0   Win=10
[2022/12/02 22:47:18] Server -> Client[ ACK ] Seq=721 Ack=721   Len=0   Win=10
[2022/12/02 22:47:18] Server -> Client[ ACK ] Seq=722 Ack=722   Len=0   Win=10
[2022/12/02 22:47:18] Server -> Client[ ACK ] Seq=723 Ack=723   Len=0   Win=10
[2022/12/02 22:47:18] [ Log ] 2.jpg transmission completed. Cost 9.908s
```


传输 3.jpg

Server

```
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2185 Ack=2185 Len=0 Win=10
[2022/12/02 22:48:04] Client -> Server[ PSH ACK ] Seq=2186 Ack=2177 Len=482 Win=8
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2186 Ack=2186 Len=0 Win=10
[2022/12/02 22:48:05] [ Log ] File 3.jpg has been written to destination folder.
```

Client

```
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2178 Ack=2178 Len=0 Win=10
[2022/12/02 22:48:04] Client -> Server[ PSH ACK ] Seq=2186 Ack=2177 Len=482 Win=8
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2179 Ack=2179 Len=0 Win=10
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2180 Ack=2180 Len=0 Win=10
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2181 Ack=2181 Len=0 Win=10
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2182 Ack=2182 Len=0 Win=10
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2183 Ack=2183 Len=0 Win=10
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2184 Ack=2184 Len=0 Win=10
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2185 Ack=2185 Len=0 Win=10
[2022/12/02 22:48:04] Server -> Client[ ACK ] Seq=2186 Ack=2186 Len=0 Win=10
[2022/12/02 22:48:04] [ Log ] 3.jpg transmission completed. Cost 14.652s
```

传输 helloworld.txt

Server

```
[2022/12/02 22:49:39] Server -> Client[ ACK ] Seq=191 Ack=191 Len=0 Win=10
[2022/12/02 22:49:39] Client -> Server[ PSH ACK ] Seq=192 Ack=186 Len=4032 Win=7
[2022/12/02 22:49:39] Server -> Client[ ACK ] Seq=192 Ack=192 Len=0 Win=10
[2022/12/02 22:49:39] [ Log ] File helloworld.txt has been written to destination folder.
```

Client

```
[2022/12/02 22:49:39] Client -> Server[ ACK ] Seq=191 Ack=184 Len=8192 Win=6
[2022/12/02 22:49:39] Client -> Server[ PSH ACK ] Seq=192 Ack=186 Len=4032 Win=7
[2022/12/02 22:49:39] Server -> Client[ ACK ] Seq=189 Ack=189 Len=0 Win=10
[2022/12/02 22:49:39] Server -> Client[ ACK ] Seq=190 Ack=190 Len=0 Win=10
[2022/12/02 22:49:39] Server -> Client[ ACK ] Seq=191 Ack=191 Len=0 Win=10
[2022/12/02 22:49:39] Server -> Client[ ACK ] Seq=192 Ack=192 Len=0 Win=10
[2022/12/02 22:49:39] [ Log ] helloworld.txt transmission completed. Cost 4.646s
```

四次挥手





Server

```
[2022/12/02 22:50:38] Receive First Handwave
[2022/12/02 22:50:38] Client -> Server[ FIN ACK ] Seq=3 Ack=2 Len=0 Win=0
[2022/12/02 22:50:38] [ Log ] Send Second Handwave
[2022/12/02 22:50:38] Server -> Client[ ACK ] Seq=3 Ack=3 Len=0 Win=0
[2022/12/02 22:50:38] [ Log ] Send Third Handwave
[2022/12/02 22:50:38] Server -> Client[ FIN ACK ] Seq=4 Ack=3 Len=0 Win=0
[2022/12/02 22:50:38] [ Log ] Receive Fourth Handwave
[2022/12/02 22:50:38] Client -> Server[ ACK ] Seq=4 Ack=4 Len=0 Win=0
```

Client

```
[2022/12/02 22:50:38] [ Log ] Send First Handwave
[2022/12/02 22:50:38] Client -> Server[ FIN ACK ] Seq=3 Ack=2 Len=0 Win=0
[2022/12/02 22:50:38] [ Log ] Receive Second Handwave
[2022/12/02 22:50:38] Server -> Client[ ACK ] Seq=3 Ack=3 Len=0 Win=0
[2022/12/02 22:50:38] Receive Third Handwave
[2022/12/02 22:50:38] Server -> Client[ FIN ACK ] Seq=4 Ack=3 Len=0 Win=0
[2022/12/02 22:50:38] Send Fourth Handwave
[2022/12/02 22:50:38] Client -> Server[ ACK ] Seq=4 Ack=4 Len=0 Win=0
```


目标文件夹

 1.jpg	2022/12/2 20:24	JPG 图片文件	3,628 KB
 2.jpg	2022/12/2 20:24	JPG 图片文件	11,521 KB
 3.jpg	2022/12/2 20:25	JPG 图片文件	81,820 KB
 helloworld.txt	2022/12/2 22:49	文本文档	6,064 KB

五、问题与分析

在本次实验的代码编写过程中主要遇到的问题主要还是接受与发送的二时延导致接收端 `cout` 日志打印被中断导致日志错位的问题，其实改用 `printf` 可以避免，但由于要修改的地方实在太多，就作罢了。

通过本次实验完成了拥塞控制 `reno` 算法的代码实现，在代码编写过程中巩固了课堂理论知识，培养了实践能力，`debug`、改进优化以及整体落实较前两次实验更为顺畅。