南開大學
Nankai University

基于 UDP 服务可靠传输协议编程实现（3-2）

姓 名：　边　笛

学 号：　2012668

学 院：计算机学院

目 录

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。流量控制采用停等机制，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）：

从低向高依次为 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);
};
```

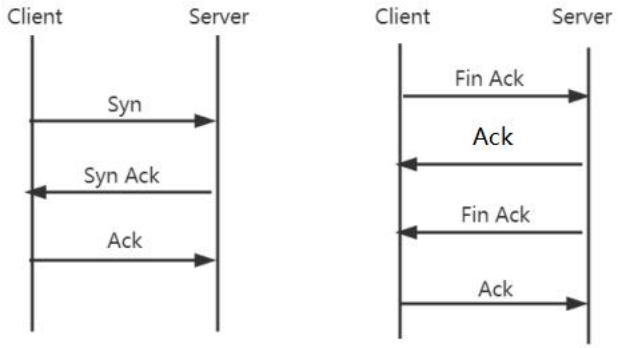
```
void setAck()
{
    Flags = Flags | 0x0010;
};
bool getAck()
{
    return Flags & 0x0010;
};
```

```
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:

```
[2022/11/29 12:31:52] [ Log ] Receive First Handshake
[2022/11/29 12:31:52] Client -> Server[ SYN ] Seq=0   Len=0   Win=10
[2022/11/29 12:31:52] [ Log ] Send Second Handshake
[2022/11/29 12:31:52] Server -> Client[ SYN  ACK ] Seq=1  Ack=0   Len=0   Win=10
[2022/11/29 12:31:52] [ Log ] Receive Third Handshake
[2022/11/29 12:31:52] Client -> Server[ ACK ] Seq=1  Ack=1   Len=0   Win=10
```

Client：

```
[2022/11/29 12:31:52] [ Log ] Send First Handshake
[2022/11/29 12:31:52] Client -> Server[ SYN ] Seq=0   Len=0   Win=10
[2022/11/29 12:31:52] [ Log ] Receive Second Handshake
[2022/11/29 12:31:52] Server -> Client[ SYN  ACK ] Seq=1  Ack=0   Len=0   Win=10
[2022/11/29 12:31:52] [ Log ] Send Third Handshake
[2022/11/29 12:31:52] Client -> Server[ ACK ] Seq=1  Ack=1   Len=0   Win=10
```

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:

```
[2022/11/29 12:33:09] Receive First Handwave
[2022/11/29 12:33:09] Client -> Server[ FIN  ACK ] Seq=3  Ack=2   Len=0   Win=10
[2022/11/29 12:33:09] [ Log ] Send Second Handwave
[2022/11/29 12:33:09] Server -> Client[ ACK ] Seq=3  Ack=3   Len=0   Win=10
[2022/11/29 12:33:09] [ Log ] Send Third Handwave
[2022/11/29 12:33:09] Server -> Client[ FIN  ACK ] Seq=4  Ack=3   Len=0   Win=10
[2022/11/29 12:33:09] [ Log ] Receive Fourth Handwave
[2022/11/29 12:33:09] Client -> Server[ ACK ] Seq=4  Ack=4   Len=0   Win=10
```

Client:

```
[2022/11/29 12:33:09] [ Log ] Send First Handwave
[2022/11/29 12:33:09] Client -> Server[ FIN  ACK ] Seq=3  Ack=2   Len=0   Win=10
[2022/11/29 12:33:09] [ Log ] Receive Second Handwave
[2022/11/29 12:33:09] Server -> Client[ ACK ] Seq=3  Ack=3   Len=0   Win=10
[2022/11/29 12:33:09] Receive Third Handwave
[2022/11/29 12:33:09] Server -> Client[ FIN  ACK ] Seq=4  Ack=3   Len=0   Win=10
[2022/11/29 12:33:09] Send Fourth Handwave
[2022/11/29 12:33:09] Client -> Server[ ACK ] Seq=4  Ack=4   Len=0   Win=10
```

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

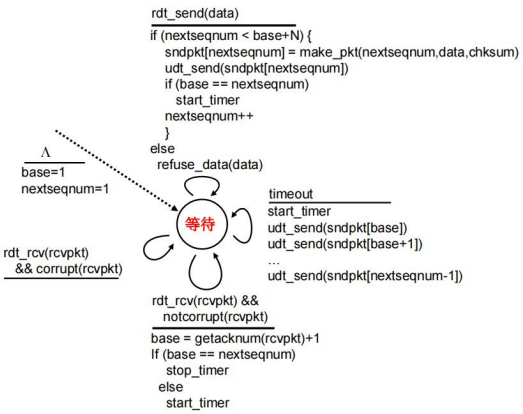
在接收到数据报后，接收端需要对数据报进行检查，对整个数据报反码求和。根据校验和的设置，可以判断当相加计算结果位全部为 1 说明没有检测到错误。基本思路如下：

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



发送端的发送不依靠于收到确认，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。

```
while(1)
{
    //窗口大小判断
    if(sendQ.size() < Max_window )
    {
        if(!k)//非尾部
        {
            struct Message sendcontent{Client_Port , Server_Port ,  lastsendSEQ+1 , lastrecvSEQ };
            sendcontent.datalen = contentlen;
            sendcontent.setAck();
            sendcontent.setData(content);
            sendcontent.Winsize = Max_window - (sendQ.size()+1);
            sendcontent.setChecksum(&sendHeader);
            bool LOSS = randomLoss();
            int s;
            if(!LOSS)
            {
                s = sendto(Client, (char*) &sendcontent, sizeof(struct Message), 0, (sockaddr*)&Server_addr, l );
            }
            else
            {
                printTime();
                cout<<"Loss Seq"<<lastsendSEQ+1<<endl;;
            }
                std::lock_guard<std::mutex> lockGuard(bufferMutex);
                // printTime();
                // std::cout<<"[ Log ] Send File Data Block "<<i+1<<endl;
                if(!LOSS) printMess(sendcontent,SEND);
                sendQ.push(sendcontent);   滑动窗维护
                lastsendSEQ = sendcontent.seq;
                filetimer = clock();
                return;
```

图 1 发送端的发送线程

窗口使用队列的数据结构来维护。每发送一条数据报，就将该条信息挂在队列里（也就是放入窗口）。由于采取的是累计确认的方式，在收到接收端发来的确认信息后，会从队列中弹出序列号小于等于接收到的 ACK 的待确认信息，相当于窗口的移动。如果超时需要重传，则重传所有已发送待确认的数据。

对于发送端接受和发送的完成需要使用到多线程。一条线程用于数据发送，另一条线程用于确认信息的接收。由于两个线程都涉及到滑动窗口的维护，因此需要对于涉及到窗口队列的操作加锁，保证正确性。

```
void Recving()
{
    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)
        {
            //接收到目标确认信息
            std::lock_guard<std::mutex> lockGuard(bufferMutex);
            printMess(recvcontent,!SEND);
            lastrecvSEQ = recvcontent.seq;
            recvWin = recvcontent.Winsize;
            //计时
            filetimer = clock();
            while(sendQ.front().seq<=recvcontent.ack)
            {
                sendQ.pop();
            }
        }
        else
        {
            //接收到错误信息，不做处理，判断超时，超时重传
            if(clock()-filetimer >= Max_waitTime)//超时
            {
                printTime();
                std::cout<<"[ Log ] Timeout! Retransmit data."<<endl;
                Resend();
            }
        }
        if(finalSEQ != 0 && recvcontent.ack == finalSEQ)
        {
            thread_switch = false;
            return;
        }
    }
}
```

图 2 发送端的接收线程

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

```
if( r!= SOCKET_ERROR && recvcontent.Check(&recvHeader) && recvcontent.Flags == 16 &&recvcontent.seq >= lastrecvSEQ +1 && recvcontent.ack <= lastsendSEQ)
{
    //接收到目标确认信息
    std::lock_guard<std::mutex> lockGuard(bufferMutex);
    printMess(recvcontent,!SEND);
    lastrecvSEQ = recvcontent.seq;
    recvWin = recvcontent.Winsize;
    //计时
    filetimer = clock();
    while(sendQ.front().seq<=recvcontent.ack)
    {
        sendQ.pop();
    }
}
```

滑动窗移动

图 3 发送端接受到确认信息，移动滑动窗

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吓一条消息，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。

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。

```
else
{
    //接收到错误信息，不做处理，判断超时，超时重传
    if(clock()-filetimer >= Max_waitTime)//超时
    {
        printTime();
        std::cout<<"[ Log ] Timeout! Retransmit data."<<endl;
        Resend();
    }
}
```

图 4 发送端重传判断

```
void Resend()
{
    std::lock_guard<std::mutex> lockGuard(bufferMutex);
    int queuesize = sendQ.size();
    //遍历队列重传
    for(int i=0;i < queuesize; i++)
    {
        struct Message resend = sendQ.front();
            int s = sendto(Client, (char*) &resend, sizeof(struct Message), 0, (sockaddr*)&Server_addr, 1 );
                // printTime();
                // std::cout<<"[ Log ] ReSend File Data Block "<<endl;
                printMess(sendQ.front(),SEND);
                lastsendSEQ = resend.seq;
                sendQ.push(resend);
                sendQ.pop();
    }
}
```

图 5 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。

7. 丢包设置

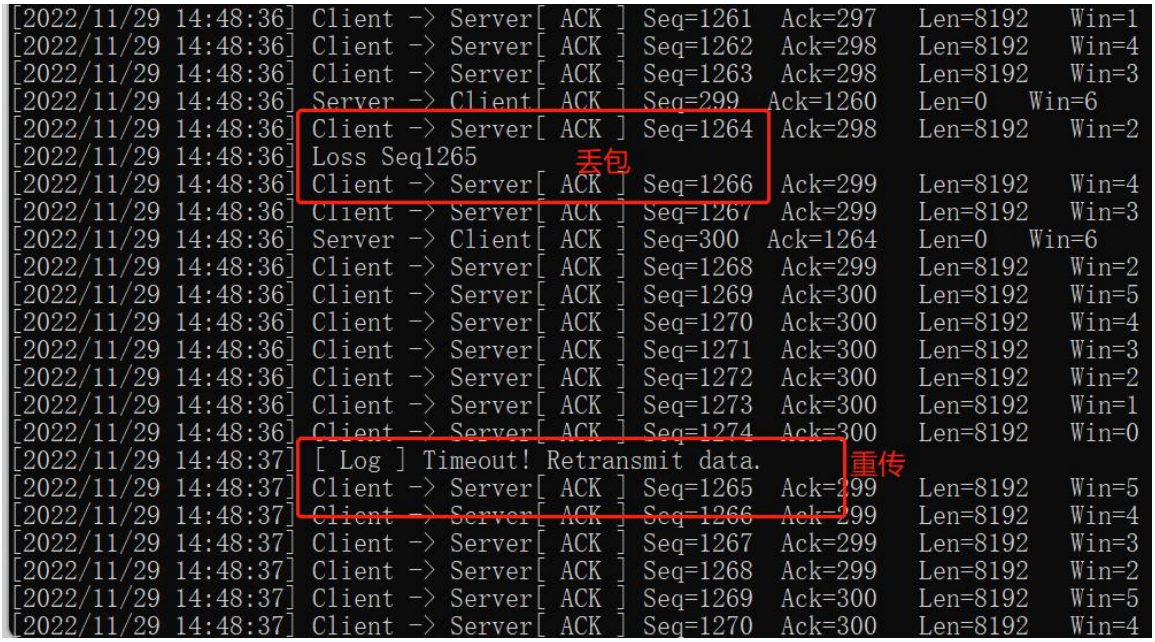
在头文件中编写了丢包函数：

```
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, l );
}
else
{
    printTime();
    cout<<"Loss Seq"<<lastsendSEQ+1<<endl;
}
```

运行程序进行检验。



8. 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。

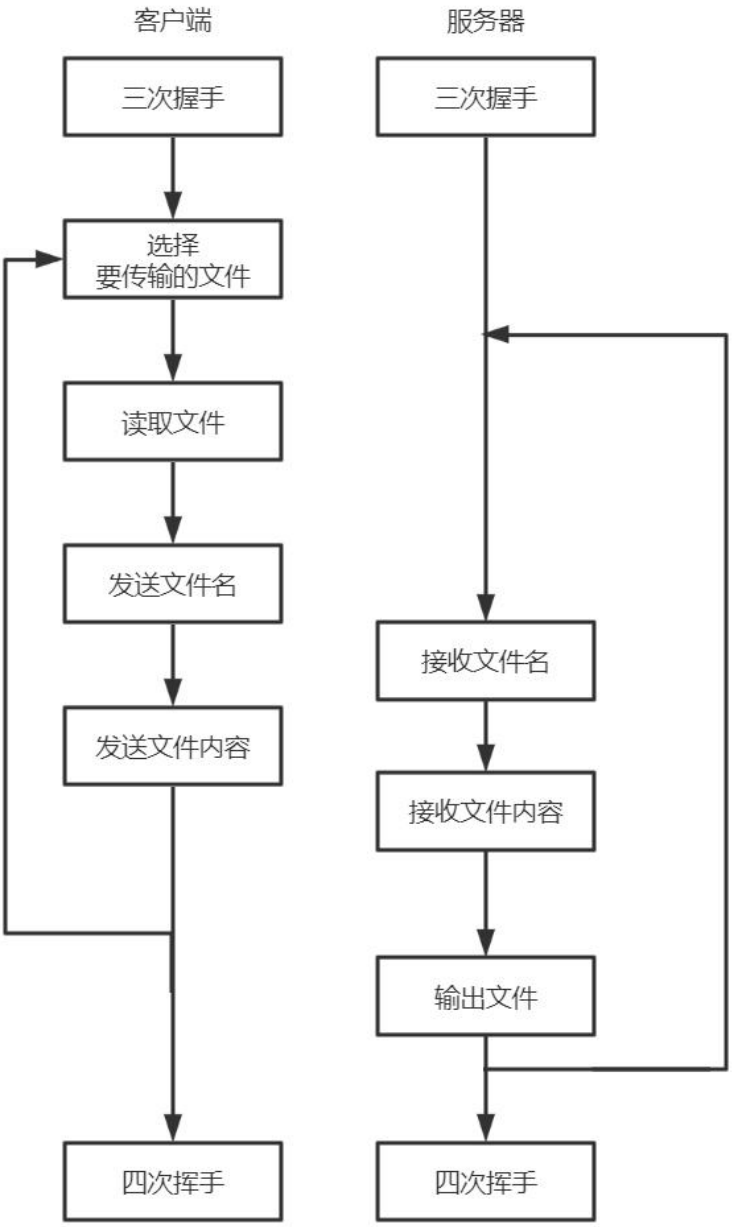
9. 日志输出

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。

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条个文件都进行了传输：

三次握手

Server

```
[2022/11/29 14:22:16] [ Log ] Receive First Handshake
[2022/11/29 14:22:16] Client -> Server[ SYN ] Seq=0   Len=0   Win=10
[2022/11/29 14:22:16] [ Log ] Send Second Handshake
[2022/11/29 14:22:16] Server -> Client[ SYN  ACK ] Seq=1  Ack=0   Len=0   Win=10
[2022/11/29 14:22:16] [ Log ] Receive Third Handshake
[2022/11/29 14:22:16] Client -> Server[ ACK ] Seq=1  Ack=1   Len=0   Win=10
```

Client

```
[2022/11/29 14:22:16] [ Log ] Send First Handshake
[2022/11/29 14:22:16] Client -> Server[ SYN ] Seq=0   Len=0   Win=10
[2022/11/29 14:22:16] [ Log ] Receive Second Handshake
[2022/11/29 14:22:16] Server -> Client[ SYN  ACK ] Seq=1  Ack=0   Len=0   Win=10
[2022/11/29 14:22:16] [ Log ] Send Third Handshake
[2022/11/29 14:22:16] Client -> Server[ ACK ] Seq=1  Ack=1   Len=0   Win=10
-------------------------Transfor---------------------------
```

传输 1.jpg

Server

```
[2022/11/29 14:46:08] Client -> Server[ ACK ] Seq=227  Ack=55   Len=8192   Win=5
[2022/11/29 14:46:09] Client -> Server[ ACK ] Seq=228  Ack=55   Len=8192   Win=4
[2022/11/29 14:46:09] Client -> Server[ PSH  ACK ] Seq=229  Ack=56   Len=5961   Win=7
[2022/11/29 14:46:09] Server -> Client[ ACK ] Seq=57  Ack=229   Len=0   Win=10
[2022/11/29 14:46:09] [ Log ] File 1.jpg has been written to destination folder.
```

Client

```
[2022/11/29 14:46:09] Client -> Server[ ACK ] Seq=227  Ack=55   Len=8192   Win=5
[2022/11/29 14:46:09] Client -> Server[ ACK ] Seq=228  Ack=55   Len=8192   Win=4
[2022/11/29 14:46:09] Client -> Server[ PSH  ACK ] Seq=229  Ack=56   Len=5961   Win=7
[2022/11/29 14:46:09] Server -> Client[ ACK ] Seq=57  Ack=229   Len=0   Win=10
```

传输 2.jpg

Server

```
[2022/11/29 14:47:44] Client -> Server[ ACK ] Seq=720  Ack=171   Len=8192   Win=4
[2022/11/29 14:47:44] Client -> Server[ ACK ] Seq=721  Ack=171   Len=8192   Win=3
[2022/11/29 14:47:44] Client -> Server[ ACK ] Seq=722  Ack=172   Len=8192   Win=6
[2022/11/29 14:47:44] Server -> Client[ ACK ] Seq=173  Ack=722   Len=0   Win=6
[2022/11/29 14:47:44] Client -> Server[ PSH  ACK ] Seq=723  Ack=172   Len=265   Win=5
[2022/11/29 14:47:44] Server -> Client[ ACK ] Seq=174  Ack=723   Len=0   Win=10
[2022/11/29 14:47:44] [ Log ] File 2.jpg has been written to destination folder.
```

Client

```
[2022/11/29 14:47:44] Client -> Server[ ACK ] Seq=721  Ack=171   Len=8192   Win=3
[2022/11/29 14:47:44] Client -> Server[ ACK ] Seq=722  Ack=172   Len=8192   Win=6
[2022/11/29 14:47:44] Client -> Server[ PSH  ACK ] Seq=723  Ack=172   Len=265   Win=5
[2022/11/29 14:47:44] Server -> Client[ ACK ] Seq=173  Ack=722   Len=0   Win=6
[2022/11/29 14:47:44] Server -> Client[ ACK ] Seq=174  Ack=723   Len=0   Win=10
```

传输 3.jpg

Server

```
[2022/11/29 14:48:49] Server -> Client[ ACK ] Seq=519  Ack=2184   Len=0    Win=6
[2022/11/29 14:48:49] Client -> Server[ ACK ] Seq=2185  Ack=517   Len=8192   Win=1
[2022/11/29 14:48:49] Client -> Server[ PSH  ACK ] Seq=2186  Ack=517   Len=482   Win=0
[2022/11/29 14:48:49] Server -> Client[ ACK ] Seq=520  Ack=2186   Len=0    Win=10
[2022/11/29 14:48:51] [ Log ] File 3.jpg has been written to destination folder.
```

Client

```
[2022/11/29 14:48:49] Client -> Server[ ACK ] Seq=2184  Ack=517   Len=8192   Win=2
[2022/11/29 14:48:49] Client -> Server[ ACK ] Seq=2185  Ack=517   Len=8192   Win=1
[2022/11/29 14:48:49] Client -> Server[ PSH  ACK ] Seq=2186  Ack=517   Len=482   Win=0
[2022/11/29 14:48:49] Server -> Client[ ACK ] Seq=518  Ack=2180   Len=0    Win=6
[2022/11/29 14:48:49] Server -> Client[ ACK ] Seq=519  Ack=2184   Len=0    Win=6
[2022/11/29 14:48:49] Server -> Client[ ACK ] Seq=520  Ack=2186   Len=0    Win=10
```

传输 helloworld.txt

Server

```
[2022/11/29 14:49:22] Server -> Client[ ACK ] Seq=566  Ack=2375   Len=0    Win=6
[2022/11/29 14:49:22] Client -> Server[ ACK ] Seq=2376  Ack=564   Len=8192   Win=1
[2022/11/29 14:49:22] Client -> Server[ PSH  ACK ] Seq=2377  Ack=564   Len=4032   Win=0
[2022/11/29 14:49:22] Server -> Client[ ACK ] Seq=567  Ack=2377   Len=0    Win=10
[2022/11/29 14:49:23] [ Log ] File helloworld.txt has been written to destination folder.
```

Client

```
[2022/11/29 14:49:22] Client -> Server[ ACK ] Seq=2376  Ack=564   Len=8192   Win=1
[2022/11/29 14:49:22] Client -> Server[ PSH  ACK ] Seq=2377  Ack=564   Len=4032   Win=0
[2022/11/29 14:49:22] Server -> Client[ ACK ] Seq=565  Ack=2371   Len=0    Win=6
[2022/11/29 14:49:22] Server -> Client[ ACK ] Seq=566  Ack=2375   Len=0    Win=6
[2022/11/29 14:49:22] Server -> Client[ ACK ] Seq=567  Ack=2377   Len=0    Win=10
```

四次挥手

Server

```
[2022/11/29 14:50:30] Receive First Handwave
[2022/11/29 14:50:30] Client -> Server[ FIN  ACK ] Seq=2379  Ack=568   Len=0    Win=10
[2022/11/29 14:50:30] [ Log ] Send Second Handwave
[2022/11/29 14:50:30] Server -> Client[ ACK ] Seq=569  Ack=2379   Len=0    Win=10
[2022/11/29 14:50:30] [ Log ] Send Third Handwave
[2022/11/29 14:50:30] Server -> Client[ FIN  ACK ] Seq=570  Ack=2379   Len=0    Win=10
[2022/11/29 14:50:30] [ Log ] Receive Fourth Handwave
[2022/11/29 14:50:30] Client -> Server[ ACK ] Seq=2380  Ack=570   Len=0    Win=10
请按任意键继续. . .
```

Client

```
[2022/11/29 14:50:30] [ Log ] Send First Handwave
[2022/11/29 14:50:30] Client -> Server[ FIN  ACK ] Seq=2379  Ack=568   Len=0    Win=10
[2022/11/29 14:50:30] [ Log ] Receive Second Handwave
[2022/11/29 14:50:30] Server -> Client[ ACK ] Seq=569  Ack=2379   Len=0    Win=10
[2022/11/29 14:50:30] Receive Third Handwave
[2022/11/29 14:50:30] Server -> Client[ FIN  ACK ] Seq=570  Ack=2379   Len=0    Win=10
[2022/11/29 14:50:30] Send Fourth Handwave
[2022/11/29 14:50:30] Client -> Server[ ACK ] Seq=2380  Ack=570   Len=0    Win=10
请按任意键继续. . .
```

目标文件夹

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有以下两点。

①同步与异步：由于发送端在接收与发送在两个线程里，都需要对队列进行操作，可能会同时进行操作，导致出现错误。尝试加锁解决，在所有涉及到队列操作的时候都加锁，但是在重发的时候发现不能把锁加的那么近，应该在进入重发的时候就加锁。由于会不会出现问题和每次程序执行的时候的实际情况有关，bug 并不一定会在运行的时候出现，可能程序正常着正常着就出问题了，调试的时候略微麻烦，对于线程的使用还需要学习。

②接收与发送的时延：由于发送与接收存在时延，可能消息的打印顺序会有些混乱。加了几处 Sleep 来尽量避免这种问题。

通过本次实验完成了一次简单的传输协议设置，在代码编写过程中发现了许多对于课堂知识掌握不牢固的地方，并且在理论落地上存在着许多困难，还需要培养好自己的实践能力，能够将理论知识应用于实践。