计算机网络实验3-3实验报告

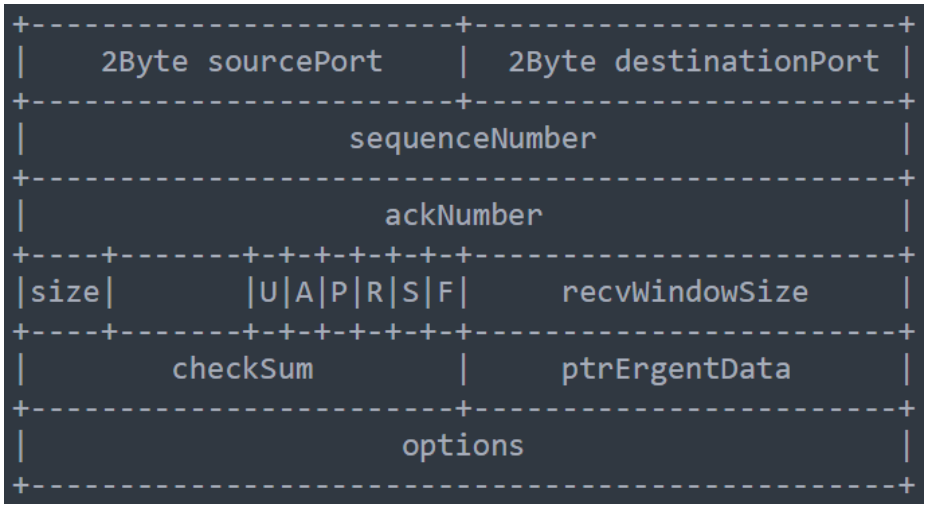
> 1911590周安琪

1 协议内容

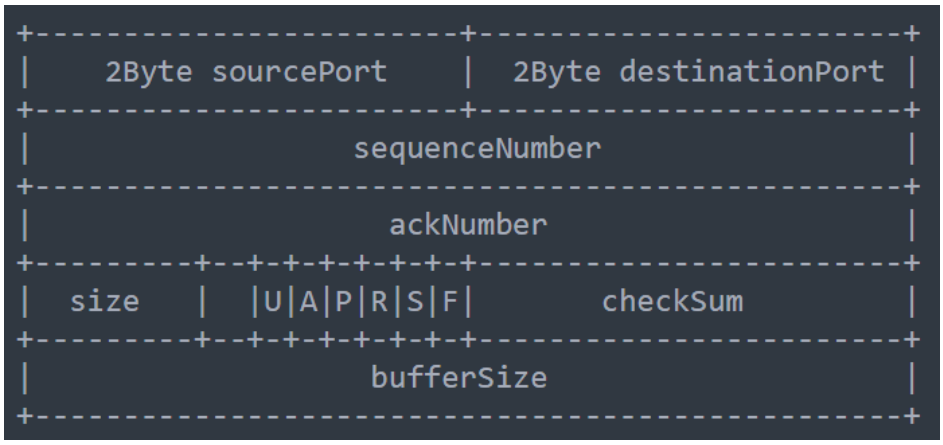
在3-2中，我实现了选择重传(SR)协议。

我参考了TCP的报文头格式设计了我的报文头。

TCP格式如下，除去非定长的options共20Byte：



在 上次实验 中，为了解决接收端最后一个包末尾全是0的问题（无法分辨写入长度），在头中加上了bufferSize，使头变成20字节大小：



在 本次实验 中，我添加了一种“请求重传报文”，在上图中 `U` 的位置，我称之为Request位。为了配合，我写了相应的setter和getter

在我实现的代码中，发送端的SYN是单独进行的。

1.1 服务端--接收端

上次实验

3-3是在上次实验的基础上进行的，上次实验中，我的逻辑如下：

当服务端被运行时，它将开始监听，一旦接收到SYN报文则发送一个SYN+ACK报文；如果报文损坏，返回空报文。同时，服务端获取接收端传来的起始序列号。

```
recvBuffer: SeqNum: 0, AckNum: 0, Size: 0, SYN: 1, ACK: 0, FIN: 0, CheckSum: 62554
Got an SYN datagram!
sendBuffer: SeqNum: 0, AckNum: 0, Size: 0, SYN: 1, ACK: 1, FIN: 0, CheckSum: 58458
Sent SYN ACK.
```

服务端的窗口大小为定长，我这里设置为16。接收到SYN报文后，服务端的窗口进行初始化，将窗口的期望序列号设置为seq~seq+16。

接着开始收到文件报文，每接收到一个报文，服务端遍历窗口判断是否匹配到窗口内的序列号。

- 如果匹配上，则把recvBuffer写到对应窗口位的buffer中去，返回对应的ack，检查是否需要移动窗口（移动窗口即为向上层应用交付接收到的内容，在本次实验中也就是向文件中写数据）
- 如果传来报文的seq在窗口最左端的左侧，也就是说，这个报文已经被ack过而且已经交付给上层了，将会对这个序列号再发一个ack。（否则发送端会一直重传）
- 如果传来报文的seq在窗口最右端的右侧，也就是说，这个报文太早了，那么不做反应。

在移动窗口的时候，每次仅仅移动一位并写回数据，移动完毕之后再检查是否还需要移动，如果还需要移动的话，将递归调用move函数。

在写回数据的时候，如果发现报文的FIN位是1，那么在写完后关闭文件。

改进

我在服务端（接收端）维护了一个数字waitingNum，意思是接收窗口中堵塞的数量，也就是：因为最左侧未接收到导致右侧都无法交付的包的数量。

当这个数字超过窗口大小的1/2时，接收端将会给发送端发送request报文，发送端接收到之后会立刻重传

1.2 客户端--发送端

上次实验

当客户端被运行时，它将向服务器端发送SYN报文，直到接收到服务器的ACK才跳出循环，开始发送文件。

```
sendBuffer: SeqNum: 0, AckNum: 0, Size: 0, SYN: 1, ACK: 0, FIN: 0, CheckSum: 62554
sent.
recvBuffer: SeqNum: 0, AckNum: 0, Size: 0, SYN: 1, ACK: 1, FIN: 0, CheckSum: 58458
Got a SYN ACK!
Tell me which file you want to send.
1.jpg
The size of this file is 1857353 bytes.
We will split this file to 30 packages and send it.
```

客户端在获取文件信息之后，会开启两个线程，设置互斥锁。两个线程一个用于发送文件，一个用于接收ack报文

- 发送文件
  - 每次拿到锁之后遍历一遍发送端的窗口
    - 如果窗口已经被ack了，不做处理
    - 如果窗口已经发送但还未被ack，检查定时器
      - 超时：将该窗口对应的buffer重传
      - 未超时：不做处理
    - 如果窗口还没被使用，则发送新的数据报文，同时开启该窗口的定时器。
  - 释放锁
- 接收ack报文
  - 每次拿到锁之后分析接收的报文
    - 如果接收到的ack序列号在窗口里，改变该窗口的ack状态，查看是否需要移动窗口。移动窗口时把最右边的窗口状态设置为“未被使用”。
    - 如果接收到的ack序列号不在窗口里，不做处理
  - 释放锁

改进

改进写在了ackReader里（也就是副线程），当ackReader接收到报文时，它将会判断是否是request报文，如果是request报文，它将会立即重传这个包。

2 开发环境

- windows10
- g++ -std=c++11

将server.cpp和client.cpp放在两个文件夹下（这是为了方便传文件，避免同名覆盖），在两个目录各运行下列指令（windows命令行）可以从cpp文件编译出exe文件：

```
g++ -std=c++11 client.cpp -o client.exe -lws2_32
g++ -std=c++11 server.cpp -o server.exe -lws2_32
```

随后使用以下指令可以分别运行两个程序：

```
client.exe
server.exe
```

3 代码解释

3.1-3.4都是上次实验的内容，此次增添的代码可以从pdf目录跳到3.5节。

3.1 client.cpp

main函数：

这里ccout是一个文件输出流，将过于详细的log输出到文件里而不是屏幕上。

主要的程序逻辑写在sendSynDatagram()和sendFileDatagram()里。其中Syn的发送和上次没区别，仅仅是添加了超时重传。

```
int main(){
    ccout.open("client.txt");

    WSADATA wsaData;
    WSAStartup(MAKEWORD(1, 1), &wsaData);

    makeSocket();
    setRTO();

    sendSynDatagram();
    sendFileDatagram();


    closesocket(sockSrv);
    WSACleanup();
    return 0;
}
```

sendSynDatagram()

随后开始发送SYN报文。

- 这里用到的 `packSynDatagram(0)` 函数是为了处理SYN报文的头信息，意思是除了SYN和SeqNum之外头都置零。传的参数是序列号之所以要传参是因为SYN报文有协商起始序列号的作用。
- `printLogSendBuffer()` 函数是为了输出日志，在后文详细展示。
- `checkSumIsRight()` 函数作用是计算校验和是否正确，即判断信息是否被损坏。

这个循环跳出的唯一条件是接收到了来自服务端的SYN+ACK报文（不可以是损坏的）。如果接收到的报文有问题，将会检测上次WSA的错误，如果是10060也就是超时错误，会重传一次报文。

```
void sendSynDatagram(){
    // 这里应该用不着滑动窗口
    while (1){
        packSynDatagram(0);
        sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
               (sockaddr*)&addrServer, len);
        printLogSendBuffer();
        ccout << "sent." << endl;

        int it=recvfrom(sockSrv, recvBuffer, sizeof(recvBuffer), 0,
                        (SOCKADDR*)&addrServer, &len);

        if(it<=0){
            if(WSAGetLastError() == 10060){// 超时处理
                printRTOErr();
                continue;
            }
        }
        else{
            printLogRecvBuffer();
            // 不断发送SYN，直到收到SYN+ACK为止。
            if(getter.getAckBit(recvBuffer) &&
               getter.getSynBit(recvBuffer) && checkSumIsRight()){
                ccout<<"Got a SYN ACK!"<<endl;
                break;
            }
        }
    }
}
```

sendGrid & sendWindow

为了实现滑动窗口我实现了两个类，窗口本身是由16个sendGrid组成的，一个sendGrid包含了：

- state：该格子的状态--0代表没被使用；1代表发了还没ack；2代表已经被ack了
- seq：该格子的序列号--一般是递增的
- buffer[BUFFER_SIZE]：该格子的缓存
- clock_t start：这个包被发送出去的时钟数，在判断该格子是否超时的时候只要获取当时的时钟数然后和它做减法就可以了。

```
class SendGrid{
public:
    int state;  //用于判断格子的状态
    int seq;    //格子的序列号是
    char buffer[BUFFER_SIZE];   //此序列号的sendbuffer
    clock_t start;//这个包被发出去的时间
    SendGrid():state(0),seq(-1){};
    void setBuffer(char* sendBuffer){
        for(int i=0;i<BUFFER_SIZE;i++){
            buffer[i]=sendBuffer[i];
        }
    }
};
```

sendWindow主要是实现了移动窗口的函数，在这里仅解说移动窗口的逻辑：

- 判断最左侧的格子是否被ack，如果被ack将会移动窗口
- 在移动窗口的时候更新“当前已经成功发送的包的数目”nowTime
- 将窗口内的所有格子做迁移-------这也是性能比较差的一个地方，用队列而不是数组性能会好很多。
- 把最右的格子设置成空闲，序列号设置为其左侧的序列号加一。
- 如果当前“已经被ack的包的数目”==“需要发送的包的数目”，那么退出程序。
- 以上做完之后，如果还是需要移动，递归调用。------递归的性能好像也不太好，可以改进。

```
// 窗口向右移动1位
void move(){
    if(sendGrid[0].state==2){//如果最左侧是不是已经ack了
        ccout<<"是时候移动窗口了！"<<endl;
        nowTime=sendGrid[0].seq+1;
        cout<<nowTime<<endl;
        for(int i=1;i<16;i++){
            sendGrid[i-1].state=sendGrid[i].state;
            sendGrid[i-1].seq=sendGrid[i].seq;
            for(int j=0;j<BUFFER_SIZE;j++){
                sendGrid[i-1].buffer[j]=sendGrid[i].buffer[j];
            }
        }
        sendGrid[15].state=0;//最右边的格子重新空闲
        sendGrid[15].seq=sendGrid[14].seq+1;
        if(nowTime==sendTimes){
            cout<<"发完了"<<endl;
            exit(0);
        }
        if(sendGrid[0].state==2)//如果最左侧还是已经ack了，继续move
            this->move();
    }
}
```

sendFileDatagram()

用于发送文件。

首先获取文件名、根据文件名读入文件，计算文件长度和“需要发送的包的数目”sendTimes。

```
getFileName();
findFile();// 根据文件名读入文件，计算文件长度和sendTimes
t_start=clock();// 用于统计
```

然后开启多线程，让ackReader开始运行。（这个函数是用于接收ack的）

```
// 用于多线程
HANDLE hThread = CreateThread(NULL, 0, ackReader, NULL, 0, NULL);
hMutex = CreateMutex(NULL, FALSE,"screen");

CloseHandle(hThread);//关闭线程的句柄，但是线程还会继续跑。
```

然后开始进入循环：

```
while(1){
    WaitForSingleObject(hMutex, INFINITE);
    //.....
    ReleaseMutex(hMutex);
}
```

循环体（解释主要在注释里。）：

- 每次拿到锁之后遍历一遍发送端的窗口
  - 如果窗口已经被ack了，不做处理
  - 如果窗口已经发送但还未被ack，检查定时器
    - 超时：将该窗口对应的buffer重传
    - 未超时：不做处理
  - 如果窗口还没被使用，则发送新的数据报文，同时开启该窗口的定时器。
- 释放锁

```
 WaitForSingleObject(hMutex, INFINITE);

 //遍历整个窗口
 for(int i=0;i<16;i++){
     //如果已经被ack了不做处理
     if(win.sendGrid[i].state==2)    continue;

     // 如果没被用上，那么开始发文件
     if(win.sendGrid[i].state==0){
```

```
            win.sendGrid[i].start=clock();//设置定时器
            sendData(i);//在i窗口发文件，是自动递增往后读的
        }

        // 已经用了但还没有ACK
        else{
            //判断是否超时
            int time=clock()-win.sendGrid[i].start;
            // clock是按照CPU算的，我这里给的超时是一秒
            if(time<1*CLOCKS_PER_SEC)   continue;
            // 超时了，重传
            else{
                printRTOErr();
                win.sendGrid[i].start=clock();//重新设置定时器
                resendData(i);//重新把第i个窗口里的buffer再发一遍
            }
        }
    }
    ReleaseMutex(hMutex);
```

ackReader()

此函数用于接收ack，整个程序就是一个循环。

```
    while(1){
        // 收到消息
        int it=recvfrom(sockSrv, recvBuffer, sizeof(recvBuffer), 0,
                        (SOCKADDR*)&addrServer, &len);

        // 校验和不对直接忽略。
        if(!checkSumIsRight()){
            ccout<<"CheckSum is wrong!"<<endl;
            continue
        }
        // 不是ack报文也忽略。
        if(getter.getAckBit(recvBuffer)==false){
            ccout<<"not an ack datagram!"<<endl;
            continue;
        }

        WaitForSingleObject(hMutex,INFINITE);
        //.....
        ReleaseMutex(hMutex);
    }
```

下面讲锁内的部分：

- 拿到锁之后看看接收到的ack是否与当前窗口中的某一格匹配

- 如果没匹配上不做处理
- 否则把该格子的状态置2，即“已经被ack”，然后查看window是否可以move.

```
WaitForSingleObject(hMutex,INFINITE);

int i=0;
for(;i<16;i++){
    // 如果匹配上了
    if(getter.getAckNum(recvBuffer)==win.sendGrid[i].seq){
        ccout<<"matched! window: "<<i;
        ccout<<"seq: "<<getter.getAckNum(recvBuffer)<<endl;
        break;
    }
}
// 说明一个都没匹配上
if(i==16){
    ccout<<"一个都没匹配上"<<endl;
    ReleaseMutex(hMutex);
    continue;
}

win.sendGrid[i].state=2;//把这个格子的状态位置2
win.move();

if(nowTime==sendTimes){
    ReleaseMutex(hMutex);
    return 0L;//表示返回的是long型的0
}
ReleaseMutex(hMutex);
```

sendData(i)

意思是在窗口第i个格子内发送，由于序列号是自动递增的，相当于已经设置好了，所以直接发送就可以了。（代码有删减）

```
//如果已经超过要发的序列号了
if(sequenceNumber≥sendTimes)   return;
//窗口置位
win.sendGrid[i].state=1;

// ...

if(isFirstPackage){
    packFirst();//加上Size的头
    // 数据段的前length个位置放文件名
    for(int j=0;j<fileName.length();j++){
        sendBuffer[HEAD_SIZE+j]=fileName[j];
    }
    // ...
```

```
    }
    else{
        packData();//没有Size的普通头
    }

    // ...
    fin.read(&sendBuffer[HEAD_SIZE + (DATA_SIZE - leftDataSize)],
    sendSize);// sendBuffer从什么地方开始读起，读多少
    // ...

    setBufferSize(sendSize);//把发送数据量的大小写在头里。
    setCheckSum();

    if(win.sendGrid[i].seq==sendTimes-1){//设置fin
        setFinBit(1);
        setCheckSum();
    }

    sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
    (sockaddr*)&addrServer, len);

    //保存到窗口的Buffer里面。
    for(int j=0;j<BUFFER_SIZE;j++){
        win.sendGrid[i].buffer[j]=sendBuffer[j];
    }

    memset(sendBuffer, 0, sizeof(sendBuffer));
```

resend(i)

重发窗口格子i的数据，只要从对应的buffer里面读出来再发一遍就可以了。

```
void resendData(int i){
    for(int j=0;j<BUFFER_SIZE;j++)
        sendBuffer[j]=win.sendGrid[i].buffer[j];

    sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
           (sockaddr*)&addrServer, len);
}

```

3.2 server.cpp

main函数：

在这里先是初始化了窗口（默认序列号是从0开始的）。主要的逻辑是写在recvDatagram()里面的。

```
    WSADATA wsaData;
    WSAStartup(MAKEWORD(1, 1), &wsaData);

    makeSocket();

    //初始化窗口
    for(int i=0;i<16;i++){
        win.sendGrid[i].seq=i;
    }

    recvDatagram();

    closesocket(sockSrv);
    WSACleanup();
    return 0;
```

move()

因为接收窗口的move和发送窗口的move不太一样所以再写一遍，因为种种原因，类名没有改，请不要在意。

接收窗口只有2个状态：已经ack的（1）和没有ack的（0）。

- 如果最左侧已经ack了，需要写回。
- 向文件中写数据，如果是第一个，要注意从文件名后面开始读
- 写完之后如果发现是FIN报文，把文件关掉，序列号重新设置成0-15
- 正常情况需要移动窗口
- 以上做完之后，查看是否需要继续move.

```
void move(){
    if(sendGrid[0].state==1){//如果最左侧已经ack了，需要写回。

        // 向文件中写数据，如果是第一个，要注意从文件名后面开始读
        if(sendGrid[0].seq==0)
            fileNameLength=getter.getSize(sendGrid[0].buffer);
        else    fileNameLength=0;

        unsigned int bufferSize=
            getter.getBufferSize(sendGrid[0].buffer);
        fout.write(&sendGrid[0].buffer[HEAD_SIZE+fileNameLength],
                   bufferSize);

        //写完之后如果发现是FIN报文，把文件关掉。
        if(getter.getFinBit(sendGrid[0].buffer)){
            fout.close();
            return;
        }

        //正常情况需要移动窗口
        for(int i=1;i<16;i++){
```

```
            sendGrid[i-1].state=sendGrid[i].state;
            sendGrid[i-1].seq=sendGrid[i].seq;
            for(int j=0;j<BUFFER_SIZE;j++){
                sendGrid[i-1].buffer[j]=sendGrid[i].buffer[j];
            }
        }
        sendGrid[15].state=0;//最右边的格子seq++
        sendGrid[15].seq=sendGrid[14].seq+1;
        for(int j=0;j<BUFFER_SIZE;j++)
            sendGrid[15].buffer[j]=0;

        //查看是否需要继续move.
        if(sendGrid[0].state==1)
            this->move();
    }
}
```

recvDatagram()

主要逻辑：

- SYN报文发SYN+ACK
- 文件报文
  - 校验和不对扔掉
  - 如果匹配上窗口内的
    - 如果是第一个包，读文件名开文件
    - 把该接收窗口格子的状态置1，把收到的报文存到该格子的buffer内
    - 根据收到的报文的序列号做ack报文的序列号，如果是fin报文，也把自己的fin置位
    - 发送ack
  - 如果是窗口左侧的
    - 根据序列号再发一次ack
  - 如果是窗口右侧的，不做处理。

代码如下：

```
void recvDatagram(){

    //建立连接，接收数据，判断数据校验和，回应报文
    while (1){
        recvfrom(sockSrv, recvBuffer, sizeof(recvBuffer), 0,
                 (SOCKADDR*)&addrClient, &len);
        printLogRecvBuffer();

        //如果是syn报文
        if(getter.getSynBit(recvBuffer)){
            //如果校验和没问题就返回一个SYN ACK报文，否则返回空报文
            if(checkSumIsRight()){/* ... */}
```

```
            else{/* ... */}
        }
        //如果是普通文件报文
        else{
            if(!checkSumIsRight())  continue;
            // (这里是默认SYN只发一遍，之后的全部都是File)


            //遍历窗口，查看发来的文件是否匹配上序列号
            int i=0;
            for(;i<16;i++){
                // 如果匹配上了

 if(getter.getSeqNum(recvBuffer)==win.sendGrid[i].seq){
                    // 如果是第一个，开文件、读文件名
                    if(getter.getSize(recvBuffer)!=0){
                        fileName="";
                        fileNameLength=getter.getSize(recvBuffer);
                        for(int
i=0;i<getter.getSize(recvBuffer);i++)
                            fileName+=recvBuffer[HEAD_SIZE+i];
                        fout.open(fileName,ios_base::out
                                | ios_base::app |
ios_base::binary);
                    }
                    win.sendGrid[i].state=1;//置位
                    for(int j=0;j<BUFFER_SIZE;j++)//存进buffer
                        win.sendGrid[i].buffer[j]=recvBuffer[j];
                    packAckDatagram(getter.getSeqNum(recvBuffer));

                    //如果收到了fin，挥手。
                    if(getter.getFinBit(recvBuffer)){
                        setFinBit(1);
                        setCheckSum();
                        ccout<<"file receiving ends."<<endl;
                    }
                    sendto(sockSrv, sendBuffer, sizeof(sendBuffer),
0,
                           (sockaddr*)&addrClient, len);
                    break;
                }
            }
            //没匹配上
            if(i==16){
                // 如果已经ack过了不在窗口里了，再发一遍
                if(getter.getSeqNum(recvBuffer)<win.sendGrid[0].seq)
{
                    packAckDatagram(getter.getSeqNum(recvBuffer));
                    if(getter.getFinBit(recvBuffer)){/* ... */}
```

```
                    sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
                           (sockaddr*)&addrClient, len);
                    continue;
                }
                // 如果在窗口右边，不做处理

 if(getter.getSeqNum(recvBuffer)>win.sendGrid[15].seq)
                    continue;
            }
            win.move();
        }
    }
}
```

3.3 握手的实现

在上文中已经提到过，在一开始的时候客户端会先给服务端发送一个SYN报文，随后服务端会给服务端发送SYN+ACK报文表明已经接收到了。

客户端：

```
// 发送SYN报文
while (1){
    packSynDatagram(0);
    sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
           (sockaddr*)&addrServer, len);
    printLogSendBuffer();
    cout << "sent." << endl;

    recvfrom(sockSrv, recvBuffer, sizeof(recvBuffer), 0,
             (SOCKADDR*)&addrServer, &len);
    printLogRecvBuffer();

    // 不断发送SYN，直到收到SYN+ACK为止。
    if(getter.getAckBit(recvBuffer) &&
       getter.getSynBit(recvBuffer) && checkSumIsRight()){
        cout<<"Got a SYN ACK!"<<endl;
        break;
    }
}
// 发送SYN报文
```

服务端：

```
// 如果是SYN报文
if(getter.getSynBit(recvBuffer)){
```

```
        cout<<"Got an SYN datagram!"<<endl;

        //如果校验和没问题就返回一个SYN ACK报文，否则返回空报文
        if(checkSumIsRight()){
            // SYN报文协商起始的序列号。
            expectedNum=getter.getSeqNum(recvBuffer);
            setAckNum(getter.getSeqNum(recvBuffer));
            packSynAckDatagram();
            sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
                   (sockaddr*)&addrClient, len);
            printLogSendBuffer();
            cout<<"Sent SYN ACK."<<endl;
        }
        else{
            packEmptyDatagram();
            sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
                   (sockaddr*)&addrClient, len);
            printLogSendBuffer();
            cout<<"sent."<<endl;
        }
    }
```

3.4 挥手的实现

在当前发送的已经是最后一个数据包的时候，客户端将会把自己的FIN置位.

服务端在收到该数据包的时候会返回一个FIN+ACK；在写回该数据包的时候，发现是FIN数据包，在写完之后会关闭文件。

客户端发送FIN：

```
if(nowTime==sendTimes){
    setFinBit(1);
    setCheckSum();
}
```

服务端接收处理FIN：

```
if(getter.getFinBit(recvBuffer)){//如果收到了fin，挥手。
    setFinBit(1);
    setCheckSum();
    ccout<<"file receiving ends."<<endl;
}
```

服务端写回FIN：

```
if(getter.getFinBit(sendGrid[0].buffer)){
    cout<<"我读完了，关文件了！"<<endl;
    fout.close();
}
```

3.5 3-3的改进

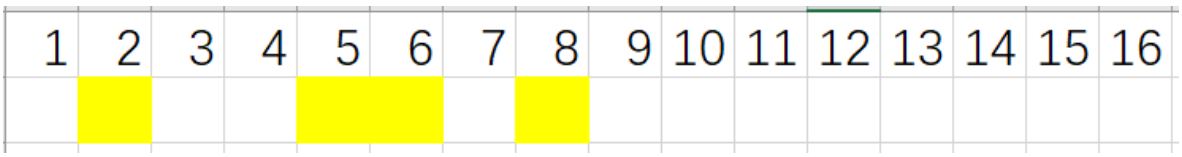
我参考了TCP的拥塞控制机制，由于可变窗口需要较大地改变数据结构，所以慢启动等较难实现。所以我主要实现了类似 3次ACK快速重传 的功能。

我在报文头里添加了request位，为此增加了setter和getter:

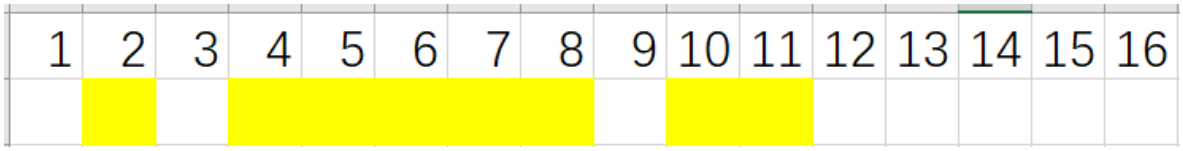
```
void setRequestBit(char a){
/*
 * +---------+--+-+-+-+-+-+-----------------------+
 * |  size   |  |R|A| | |S|F|       checkSum        |
 * +---------+--+-+-+-+-+-+-----------------------+
 */
    if(a==0){
        // 1101 1111
        sendBuffer[13] &=0x7f;
    }
    else{
        // 0010 0000
        sendBuffer[13] |=0x20;
    }
}
int getRequestBit(char* recvBuffer){
    // 0010 0000
    int a=(recvBuffer[13] & 0x20)>>5;
    return a;
}
```

server.cpp

我在接收端内维护了一个数字waitingNum，意思是窗口内有多少等待交付的内容，比如下图waitingNum=4。



当waitingNum>=WINDOW_SIZE/2，在这里是waitingNum>=8的时候，接收端将会认为网络不佳给传输带来了比较严重的影响，它将会给发送端发送一个request报文，附带请求序列号，序列号写在seqNum字段内（普通ACK报文使用的是ackNum字段）。



主要对代码做的改动如下：

- 第一次接到某个序列号的包的时候将waitingNum++
- 每次接到新的包的时候查看waitingNum是否超过半数，如果是，则发送request报文

```
if(!checkSumIsRight()) continue;
int i=0;
//遍历整个接收窗口，看接收的包和它匹不匹配
for(;i<WINDOW_SIZE;i++){
    if(getter.getSeqNum(recvBuffer)≠win.sendGrid[i].seq) continue;
    if(win.sendGrid[i].state==0){
        if(getter.getSize(recvBuffer)≠0){ ... }

        win.sendGrid[i].state=1;//改变状态
        for(int j=0;j<BUFFER_SIZE;j++)
            win.sendGrid[i].buffer[j]=recvBuffer[j];

        //#####################################################
        waitingNum++;
        cout<<"waitingNum: "<<waitingNum<<"\n";

        //#####################################################
    }
    //#####################################################
    if(waitingNum≥WINDOW_SIZE/2){
        setRequestBit(1);
        setSeqNum(win.sendGrid[0].seq);
        setCheckSum();
        sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
               (sockaddr*)&addrClient, len);
        setRequestBit(0);
        setSeqNum(0);
        cout<<"已经请求重传"<<win.sendGrid[0].seq<<"了\n";
    }
    //#####################################################
    packAckDatagram(getter.getSeqNum(recvBuffer));
    if(getter.getFinBit(recvBuffer)){//如果收到了fin，挥手。
        setFinBit(1);
        setCheckSum();
    }
    sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
           (sockaddr*)&addrClient, len);
    break;
}
//没匹配上
if(i==WINDOW_SIZE){ ... }
```

```
42    win.move();
```

client.cpp

我在接收端写了两个线程，在上个实验中，ackReader线程的作用只是接收ack并且设置窗口状态，在本次实验中，我为它增加了一个功能，即在受到request报文的时候立刻重传。

其中resendData(i)函数是直接重传窗口i所带的buffer。

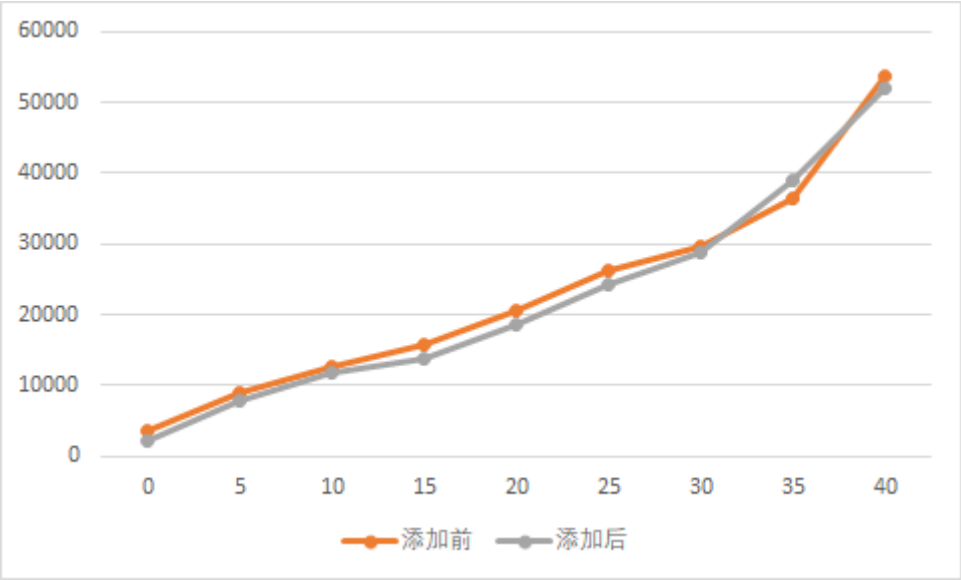
代码如下：

```
1   WaitForSingleObject(hMutex,INFINITE);
2   if(getter.getRequestBit(recvBuffer)){
3       cout<<"对方请求重传！";
4       int seq=getter.getSeqNum(recvBuffer);
5       cout<<seq<<"号"<<endl;
6       int j=0;
7       for(;j<WINDOW_SIZE;j++){
8           if(seq==win.sendGrid[j].seq){
9               cout<<"匹配上了！"<<endl;
10              resendData(j);
11              break;
12          }
13      }
14      ReleaseMutex(hMutex);
15      continue;
16  }
17   ...
18  ReleaseMutex(hMutex);
```

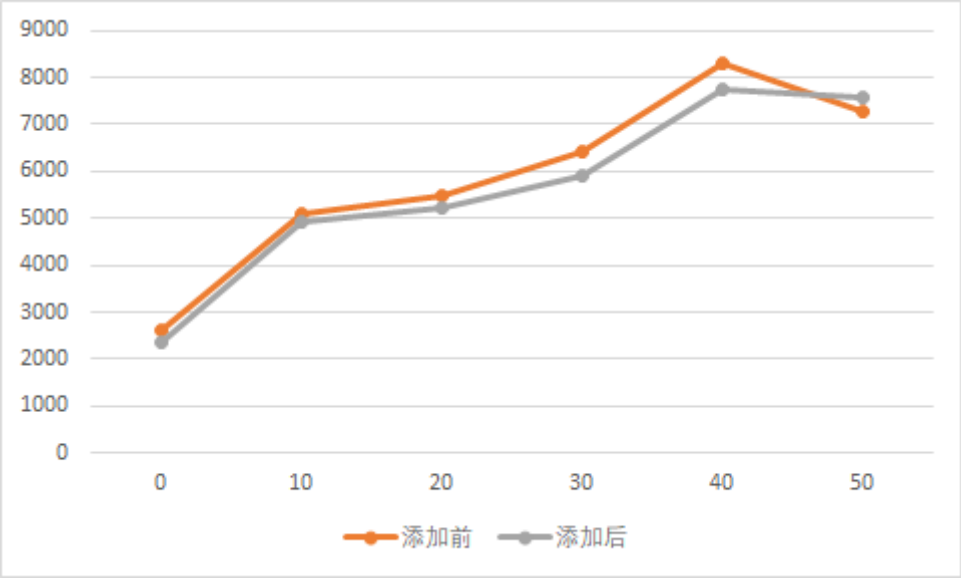
4 实验输出和对比

由于3-1我只是实现了停等机制，所以在丢包的情况下其实会陷入死循环，所以不参与对比，这里只对3-2和3-3做对比。在3-4，我会给3-1增添超时重传然后再做全面的对比。

按时延：



按丢包率：



可以看出有微弱的改善。

图和表在文件夹的`统计表.xlsx`内，此次是手动统计excel画图，下次时间将会用python绘制。

5 代码的不足之处

- 实现的拥塞控制算法非常简单，这也是受整个代码结构不够灵活所限——之前设计代码的时候没有想到它不好改动成可变窗口。
- 顺带一提，上次实验我设置超时重传的时间是1s，这次我改成0.1s后速度快了十倍。

6 相比上次实验做的改进

- 增加了快速重传功能