# 实验3：基于UDP服务设计可靠传输协议并编程实现（3-3）

## 协议设计

实现UDP协议下一个发送方对应一个接收方，

#### 发送方和接收方的主要功能：

发送方--发送文件-->接收方，接收方--确认号-->发送方

#### 发送方：

将发送与接收放入两个线程中

发送线程：

实现连续发送滑动窗口大小个数据包，存入发送缓存区senbuf，中间不需要接收确认包；如果seq超出滑动窗口，发送线程休眠，唤醒接收线程，如果为最后一个数据包，发送完后唤醒接收线程

接收线程：

传输过程中超时的话就重发发送窗口中最早发送但还没有接收到ACK确认包的那一个数据包（timeout\_retry\_for函数），{PS:建立或关闭连接超时的话重发最近一次发送的数据包(timeout\_retry函数)}判断接收到的确认包的确认号：

1. 如果收到了接收端的ACK\_FIN，表示接收端接收成功，此时不再确认剩下还未确认的数据包，直接进行关闭连接
2. 如果最后一个数据包已经发送完成，判断此时发送窗口内的数据包是否都已经进行确认，如果都进行确认，就进行关闭连接，否则就重发窗口内还未被确认的数据报
3. 如果接收到窗口下界的ACK确认包，就判断窗口内从下界开始已确认的连续数据包，更新发送窗口，并唤醒发送线程，发送更新后未发送的数据包
4. 如果收到窗口上界的ACK确认包，检查发送窗口内是否还有未确认的数据包，重新发送这些未确认的数据包
5. 如果接收到窗口范围内乱序的ACK确认包（即不是下界），标记对应的数据包为已确认（flag[i]=1）

#### 接收方：

循环接收：

1. 检查校验和
2. 检查接收数据包的seq：
   1. 如果在接收窗口范围内
      1. 如果接收到窗口下界：
         1. 判断是否是顺序接收到的(recvpdu.seq > lastrecvseq{最近接收到的最大seq数据包})，不需要缓存，直接写入文件,发送对应的ACK确认包
         2. 如果接收到的是原先丢失的下界数据包（recvpdu.seq <= lastrecvseq），就将数据包保存在接收窗口缓存区recvbuf[i]，设置接收标志bufflag[i]=1,{i= (recvpdu.seq / (nsize + 1)) % recvpdu.win}，发送对应的ACK确认包并判断接收窗口缓冲区内从窗口下界开始的连续已接收的数据包，写入文件，清空置位标志，更新接收窗口范围，如果写入了传输文件最后一个数据包，就退出接收，进行关闭连接
      2. 如果是乱序接收(收到的不是窗口下界)
         1. 就将接收到的数据包保存在对应的接收窗口缓冲区，recvbuf[i]，设置接收标志bufflag[i]=1,{i= (recvpdu.seq / (nsize + 1)) % recvpdu.win}发送对应的ACK确认包
   2. 如果小于接收窗口下界，说明是ACK确认包丢失，重传对应的ACK确认包

### 数据报格式

//每次传送的大小

#define nsize 6000

struct PDU {

uint32\_t sourceip;//源ip

uint32\_t destip;//目的ip

uint16\_t source\_port;//源端口

uint16\_t dest\_port;//目的端口

uint32\_t ack = -1;//确认号

uint32\_t seq = 0;//序列号

uint8\_t flags = 0;//标志位 ACK SYN FIN LOSS

int B\_E = -1;//0开始，1结束

uint32\_t win = 1;//滑动窗口大小

size\_t length = 0;//数据大小，保持文件大小不变

uint16\_t checksum = 0;//16位校验和

char buf[nsize] = { 0 };

};

其中，seq为发送的数据的首个字节序号，接收到ack=n表示对与字节序号为n-1之前的都进行正确的接收

### 全局变量

### 发送方：

static PDU\* sendbuf;//缓存滑动窗口大小的已经发送的数据包，方便丢包重发

static int\* bufflag;//对应sendbuf的判断，防止发送未填充的数据包

static int\* flag;//对应sendbuf中数据包的是否已确认的判断

### int lastsendbufseq;//记录最近一次从文件读入数据的seq

//窗口发送的数据报区间[start\_index, end\_index]Bytes

static uint32\_t start\_index=0;

### static uint32\_t end\_index=start\_index+(sendpdu.win-1)\*(nsize+1);

### 接收方：

static PDU\* recvbuf;//乱序接收时缓存数据包的接收窗口缓存区

static int\* bufflag;//判断recvbuf中是否接收到了对应的数据包

### int lastrecvseq=0;//最近最大接收到的seq，用于判断接收到的下界数据包是顺序到达还是乱序到达

//接受窗口区间

static uint32\_t start\_index = 0;

### static uint32\_t end\_index = start\_index + (sendpdu.win - 1) \* (nsize + 1);

### 建立连接

#### 双方

发送方和接受方采用一样的模式：

##### 初始化： 进行bind过程，绑定ip和端口号

##### 与路由器的连接：在三次握手后建立连接

##### 由于接收发窗口大小一致，故在建立连接时，如果发现窗口大小不一致，更改接收方窗口为发送方窗口大小：

##### 

#### 三次握手建立连接

1. 发送方向接收方发送SYN+(seq=0)包，请求建立连接：

如果没有在超时时间内接收到接收端的SYN+ACK+(ack=1)，会进行重传

1. 接收方收到发送方的SYN包之后，向发送方发送SYN+ACK+(ack=1)+(seq=0)包,并期望接收到发送方的ACK+（ack=1）包；

如果没有在超时时间内接收到期望的ACK就会进行重传,如果超过最长等待时间20s说明期望的ACK包应该丢失了，不再等待进入建立连接

如果在限制时间内接收到了，接收方就进入了连接建立阶段

1. 发送方接收到接收方的SYN包之后，就会向接收方发送ACK+(ack=1)+(seq=1)包,进入连接建立阶段

### 数据传输

#### 超时重传

通过setsockopt中的SO\_RCVTIMEO设置超时,初始超时时间为5ms，在循环接收时判断超时(recvret == -1 && GetLastError() == 10060) ,超时就将超时时间更新为原来的2倍，如果收到了数据，就将超时时间更新为原来的1/2，在实验过程中，发现超时时间一般最大到1024ms，故当超时时间减少到1024ms时就不再继续减少。

//设置接收超时,初始化为5ms，设置成功后，recv（）非阻塞

int timeOut = 5;

setsockopt(Socket, SOL\_SOCKET, SO\_RCVTIMEO, (char\*)&timeOut, sizeof(timeOut)

增加了timeout\_retry\_for函数；用于在传输过程中的超时重传，会重传发送窗口内最早发送但还没有收到ACK确认包的一个数据包

//传输过程中超时重传会找到重传窗口内最早没有收到ACK的包，只重传该包

int timeout\_retry\_for(PDU sendpdu, uint32\_t& start\_index, uint32\_t& end\_index) {

while (1) {

while (sendret < 0) {

cout << "send error" << GetLastError() << endl;

sendret = send(sendpdu);

cout << "=================================================resend====================================================" << endl;

pdu\_to\_str(sendpdu);

}

recvret = recv();

if (recvret > 0) {

cout << "===================================================recv===================================================" << endl;

pdu\_to\_str(recvpdu);

lessouttime();

cout << "success recv ,so less timeout" << timeOut << endl;

break;

}

else if (recvret == -1 && GetLastError() == 10060) {

int start = (start\_index / (nsize + 1)) % sendpdu.win;

int end = (end\_index / (nsize + 1)) % sendpdu.win;

if (start <= end)

for (int i = start; i <= end; i++) {

if (flag[i] == 0) {

cout << "=========Time\_Out\_Retry\_So\_Find\_The\_First\_Not\_Recv\_ACK\_Resend\_This\_PDU============seq[" << i << "]=>" << sendbuf[i].seq << "=+>" << flag[i] << endl;

send(sendbuf[i]);

pdu\_to\_str(sendbuf[i]);

break;

}

else {

cout << "ACK\_seq[" << i << "]=>" << sendbuf[i].seq << endl;

}

}

else {

int t = 0;

for (int i = start; i < sendpdu.win; i++) {

if (flag[i] == 0) {//找到了

cout << "=========Time\_Out\_Retry\_So\_Find\_The\_First\_Not\_Recv\_ACK\_Resend\_This\_PDU============seq[" << i << "]=>" << sendbuf[i].seq << "=+>" << flag[i] << endl;

send(sendbuf[i]);

pdu\_to\_str(sendbuf[i]);

t = 1;

break;

}

else {

cout << "ACK\_seq[" << i << "]=>" << sendbuf[i].seq << "=+>" << flag[i] << endl;

}

}

//还没找到，接着找

if(t==0)

for (int i = 0; i <= end; i++) {

if (flag[i] == 0) {

cout << "=========Time\_Out\_Retry\_So\_Find\_The\_First\_Not\_Recv\_ACK\_Resend\_This\_PDU===========seq[" << i << "]=>" << sendbuf[i].seq << "=+>" << flag[i] << endl;

send(sendbuf[i]);

pdu\_to\_str(sendbuf[i]);

break;

}

else {

cout << "ACK\_seq[" << i << "]=>" << sendbuf[i].seq << endl;

}

}

}

updateouttime();

}

else {

cout << "other recv error" << GetLastError() << endl;

return 0;

}

}

return 1;

##### }

##### 选择确认（窗口内的数据包或者ACK确认包丢失，只重新发送窗口内对应的数据包，超时的话会发送最早发送但还没接收到ACK的那一个数据包）

## 对接收方：

##### 更新接收窗口的时机：

##### 当接收到窗口下界，

##### 若是乱序到达：就会在接收窗口缓存区保存该包，设置接收标志位，判断从窗口下界开始连续的数据包，写入文件，清空接收标志，发送ACK确认包并更新接收窗口范围;

##### 若是顺序到达，就直接写入文件，发送ACK确认包，更新接收窗口，不再进行保存

##### 重发ACK确认包的时机：

##### 收到小于接收窗口下界的数据包，说明是确认包丢失，重发对应的确认包

## 对发送方：

##### 采用notify\_wait进行线程的唤醒和切换{当发送线程超出滑动窗口，就进入阻塞，等待接收线程完成ACK确认并更新扩大滑动窗口后唤醒发送线程继续发送}

##### threadrecv线程接收ACK包

##### 先将建立连接时多余的数据包头丢弃

##### 更新发送窗口的时机：

##### 当接收到窗口下界，将发送缓冲区从下界开始已经收到确认包的连续数据包移出发送窗口，更新发送窗口(起始位为sendbuf中第一个未收到确认包的数据包处)，唤醒发送线程，发送发送窗口内新增的可发送的数据包，发送完后唤醒接收线程

##### 检查ACK确认包接收情况的时机：

##### 当接收到窗口上界，就判断发送窗口内是否有未确认的数据包，重新发送对应的还未进行确认的数据包

##### 接收ACK确认包超时会重传发送窗口内最早发送但还没有接收到ACK确认包的数据包

threadsend线程发送数据包：

如果发送的序列号位于滑动窗口覆盖的字节序范围内就从文件读入固定大小的数据，写入sendbuf，设置sendpdu的开始结束标志，ACK位，并存入对应的sendbuf，设置对应的bufflag位置1，发送数据包

否则，将阻塞发送线程，唤醒接收线程

### 断开连接

#### 四次挥手断开连接（以发送方先关闭连接为例，Ps:接收方也可按相同步骤先关闭连接）

1. 发送方发送完数据并正确接收到最后的ACK包后，开始断开连接，向接收方发送ACK+FIN包，并期望接收到接收方的ACK包，若超时还未接收到就重发ACK+FIN
2. 接收判断文件接收完毕后，若收到ACK+FIN包，向发送方发送ACK确认包，之后继续发送ACK\_FIN包，并期望得到发送方的ACK包，若超时还未收到就会重发ACK+FIN,若超时时间超过20s,说明期望ACK丢失，不再等待，直接关闭连接，因为数据在开始关闭前已经传送完毕，最后的ACK确认包接收不影响
3. 发送方收到ACK后，判断确认号是否正确，发送方继续收到ACK\_FIN包后，向接收方发送ACK之后等待10秒后就断开连接
4. 接收方收到期望的ACK包后，断开连接

## 核心代码分析

功能函数

16位校验和 计算：

（将结构体中sourceip,destip,source\_port,dest\_port,seq,ack,length,buf，flags按每十六位（其中sourceip,destip,seq，ack分为两个十六位，length，flags零扩展至十六位，buf每两个字节当成一个十六位，多出来的一个字节零扩展至十六位）进行相加，溢出加到结果最低位） 最后结果（取低16位后再取反）放入checksum中

uint16\_t setchecksum(PDU& sendpdu) {

sendpdu.checksum = 0;

uint32\_t tem = add(add(add(add(add(add(add(sendpdu.source\_port, sendpdu.dest\_port), sendpdu.ack), sendpdu.seq), sendpdu.flags), sendpdu.length),sendpdu.sourceip),sendpdu.destip);

int i = 0;

for (int i = 0; i < 6000; i += 2) {

uint16\_t t = 0 | sendpdu.buf[i] | sendpdu.buf[i + 1] << 8;

tem = add(tem, t);

}

while (i < 6000) {

tem = add(tem, sendpdu.buf[i]);

i++;

}

sendpdu.checksum = (~tem) & 0xffff;

return sendpdu.checksum;

}

检验：

//在计算的基础上加上checksum，溢出加到最低位，正确的检验结果应该为0xffff

int checkchecksum(PDU& recvpdu) {

uint32\_t tem = add(add(add(add(add(add(add(recvpdu.source\_port, recvpdu.dest\_port), recvpdu.ack), recvpdu.seq), recvpdu.flags), recvpdu.length),recvpdu.sourceip),recvpdu.destip);

int i = 0;

for (int i = 0; i < 6000; i += 2) {

uint16\_t t = 0 | recvpdu.buf[i] | recvpdu.buf[i + 1] << 8;

tem = add(tem, t);

}

while (i < 6000) {

tem = add(tem, recvpdu.buf[i]);

i++;

}

tem = tem + recvpdu.checksum;

if (tem == 0xffff) { return 1; }

else { return 0; }

}

1. 更新滑动窗口覆盖的字节序列：

//更新发送区间

void update\_winval(uint32\_t&start\_index, uint32\_t&end\_index,int winsize, uint32\_t nextsendseq) {

cout << "update\_window\_from: [" << start\_index << "," << end\_index << "] to";

start\_index = nextsendseq;

end\_index = start\_index + ((sendpdu.win-1) \* (nsize + 1));

cout<<"[" <<start\_index<<","<<end\_index<<"]"<<endl;}

1. 超时重传 由于设置setsocketopt中接收超时，故recv()非阻塞，需要循环接收

int timeout\_retry(PDU &sendpdu) {

while (1) {

while (sendret < 0) {

cout << "send error" << GetLastError() << endl;

sendret = send(sendpdu);

cout << "======================resend=======================" << endl;

pdu\_to\_str(sendpdu);

}

recvret = recv();

cout << "======================recv=======================" << endl;

pdu\_to\_str(recvpdu);

if (recvret > 0) {

cout << "================recv============================:" << endl;

pdu\_to\_str(recvpdu);

lessouttime();

cout << "success recv ,so less timeout" << timeOut << endl;

break;

}

else if (recvret == -1 && GetLastError() == 10060) {

sendret = send(sendpdu);

cout << "======================resend=======================" << endl;

pdu\_to\_str(sendpdu);

updateouttime();

cout << "timeout happen,so more timeout" << timeOut << endl;

if (timeOut > 20000) {

//时间太长了,不继续等待了

return 0;

}

}

else {

cout << "other recv error" << GetLastError() << endl;

return 0;

}

}

return 1;

}

1. 述描述的三次握手建立连接一致

//发送方

bool establish\_conn() {

//第一次握手

sendpdu.flags = SYN;

sendpdu.seq = 0;

sendret = send(sendpdu);

if (sendret < 0) {

cout << "send error" << GetLastError() << endl;

return false;

}

//第二次握手

timeout\_retry(sendpdu);

cout << "======================send=======================" << endl;

pdu\_to\_str(sendpdu);

while (!((recvpdu.ack == sendpdu.seq + 1))) {

sendret = send(sendpdu);

timeout\_retry(sendpdu);

}

{

cout << "SECOND SHAKE HAND SUCCEED！！" << endl;

cout << "======================recv=======================" << endl;

pdu\_to\_str(recvpdu);

}

//第三次握手

sendpdu.flags = ACK;

sendpdu.seq++;

sendpdu.ack = recvpdu.seq + 1;

sendret = send(sendpdu);

cout << "======================send=======================" << endl;

pdu\_to\_str(sendpdu);

if (sendret > 0) {

cout << "client\_conn\_established";

return true;

}

else cout << "conn\_send\_error" << endl;

return false;

}

//接收方

bool establish\_conn() {

while (1) {

recvret = recv();

if (recv() > 0)break;

else cout << "waiting for client request......." << " " << recvret << endl;

}

sendpdu.dest\_port = ntohs(RemoteAddr.sin\_port);

cout << "======================recv=======================" << endl;

pdu\_to\_str(recvpdu);

if (recvpdu.win != sendpdu.win) {//修改窗口大小一致=发送方窗口大小

sendpdu.win = recvpdu.win;

cout << "WARNING::++++++++++++++++++++++sendwin!=recvwin,So update recvwin,let recvwin = sendwin+++++++++++++++++++" << endl;

}

if (recvpdu.flags == SYN) {

cout << "FIRST SHACK HANDS SUCCEED!!" << endl;

sendpdu.flags = ACK\_SYN;

sendpdu.seq = 0;

sendpdu.ack = recvpdu.seq + 1;

sendret = send();

//重传

timeout\_retry();

//第三次握手

while (!(recvpdu.ack == sendpdu.seq + 1 && recvpdu.flags == ACK)) {

sendret = send();

if (timeout\_retry() == 0) {

cout << "THRID SHACK HAND NOT ARRVE!!!no try anymore,establish conn" << endl;

return true;

}

}

{

cout << "THRID SHACK HAND SUCCEED!!!" << endl;

cout << "======================recv=======================" << endl;

pdu\_to\_str(recvpdu);

cout << "server\_conn\_established"; return true;

}

return false;

}

return false;

}

1. 释放连接,与上述描述的四次挥手释放连接一致

//发送方

void close\_conn() {//先关闭连接

sendpdu.flags = ACK\_FIN;

sendpdu.seq = 0;

//超时重传

sendret = send(sendpdu);

cout << "======================send=========================" << endl;

pdu\_to\_str(sendpdu);

cout << "FIRST BYE，go state: FIN\_WAIT1" << endl;

while (recvpdu.ack != 1) {

sendret = send(sendpdu);

}

//timeout\_retry(sendpdu);

cout << "go state FIN\_WAIT2" << endl;

while (!(recvpdu.ack == sendpdu.seq + 1 && recvpdu.flags == ACK\_FIN)) {

Sleep(1000);

timeout\_retry(sendpdu);

}

sendpdu.flags = ACK;

sendpdu.ack = recvpdu.seq + 1;

sendret = send(sendpdu);

sendret = send(sendpdu);

cout << "======================send=========================" << endl;

pdu\_to\_str(sendpdu);

cout << "THIRD BYE ,go state:TIME\_WAIT" << endl;

Sleep(10000);//等待10秒

closesocket(Socket);

}

bool close\_conn\_after() {//后关闭连接

//函数前收到了对端的ACK\_FIN

sendpdu.flags = ACK;

sendpdu.ack = recvpdu.seq + 1;

cout << "SECOND BYE,go to state CLOSE\_WAIT" << endl;

sendret = send(sendpdu);

//已经接收完毕,发送ACK+FIN

cout << "======================send=========================" << endl;

pdu\_to\_str(sendpdu);

sendpdu.flags = ACK\_FIN;

sendpdu.seq = 0;

sendret = send(sendpdu);

cout << "======================send=========================" << endl;

pdu\_to\_str(sendpdu);

cout << "THIRD BYE,go to state LAST\_ACK" << endl;

//超时重传

if (timeout\_retry(sendpdu) == 0) {

cout << "FOUR BYES NOT ARRIVED but not wait anymore,close connect!!" << endl;

closesocket(Socket);

return true;

}

if (check\_ack\_seq(sendpdu, recvpdu)) {

cout << "FOUR BYES SUCCEED!!" << endl;

closesocket(Socket);

return true;

}

return false;

}

//接收方

void close\_conn\_first() {//先关闭连接

sendpdu.flags = ACK\_FIN;

sendpdu.seq = 0;

//超时重传

sendret = send();

cout << "======================send=========================" << endl;

pdu\_to\_str(sendpdu);

cout << "FIRST BYE，go state: FIN\_WAIT1" << endl;

while (recvpdu.ack != 1&&recvpdu.seq!=0) {//没收到关闭连接对应的seq和ack（由于还有没确认的ACK），就一直发送ACK\_FIN

sendret = send();

} //timeout\_retry();

cout << "go state FIN\_WAIT2" << endl;

while (!(recvpdu.ack == sendpdu.seq + 1 && recvpdu.flags == ACK\_FIN)) {

Sleep(1000);

timeout\_retry();

}

sendpdu.flags = ACK;

sendpdu.ack = recvpdu.seq + 1;

sendret = send();

sendret = send();

cout << "======================send=========================" << endl;

pdu\_to\_str(sendpdu);

cout << "THIRD BYE ,go state:TIME\_WAIT" << endl;

Sleep(10000);//等待10秒

closesocket(Socket);

}

bool close\_conn() {//后关闭连接

//函数前收到了对端的ACK\_FIN

sendpdu.flags = ACK;

sendpdu.ack = recvpdu.seq + 1;

cout << "SECOND BYE,go to state CLOSE\_WAIT" << endl;

sendret = send();

//已经接收完毕,发送ACK+FIN

cout << "======================send=========================" << endl;

pdu\_to\_str(sendpdu);

sendpdu.flags = ACK\_FIN;

sendpdu.seq = 0;

sendret = send();

cout << "======================send=========================" << endl;

pdu\_to\_str(sendpdu);

cout << "THIRD BYE,go to state LAST\_ACK" << endl;

//超时重传

if (timeout\_retry() == 0) {

cout << "FOUR BYES NOT ARRIVED but not wait anymore,close connect!!" << endl;

closesocket(Socket);

return true;

}

if (check\_ack\_seq(sendpdu, recvpdu)) {

cout << "FOUR BYES SUCCEED!!" << endl;

closesocket(Socket);

return true;

}

return false;

}

1. Main函数

**//发送方：**

发送线程：及时发送在滑动窗口覆盖的区间范围内的数据包

int first = 0;

void threadsend(ifstream& fin, uint32\_t& start\_index, uint32\_t& end\_index, int& st) {

while (!fin.eof()) {

//接着读取文件

cout << "===sendThrea\_go\_on===";

std::unique\_lock<std::mutex> lck(mts);

cout << "滑动区间:[" << start\_index << "," << end\_index << "]" << endl;

if (!((sendpdu.seq == start\_index || sendpdu.seq > start\_index) && (sendpdu.seq == end\_index || sendpdu.seq < end\_index))) {

cout << sendpdu.seq << "未在发送区间" << start\_index << "," << end\_index << "sendThrea\_wait" << endl;

first = 1;

cv.notify\_one();

cv.wait(lck);

continue;

}

memset(sendpdu.buf, 0, sizeof(char) \* nsize);

fin.read(sendpdu.buf, nsize);//设置发送文件到sendbuf

st++;//判断是否是第一次读取（文件头)

cout << "st===" << st << endl;

//设置开始结束标志

if (st == 1) {

cout << "开始标志设置" << endl;

sendpdu.B\_E = 0; sendpdu.seq = 0;

}//第一次读取，设置开始标志和seq

else if (!fin.eof()) {

sendpdu.B\_E = -1;

}//不是第一次读取

else if (fin.eof()) {

cout << "设置结束标志" << endl;

sendpdu.B\_E = 1;

sign = 1;

}//最后一次读取，设置结束标志

lastsendbufseq = sendpdu.seq;

sendpdu.flags = ACK;//设置发送ACK

sendpdu.length = fin.gcount();

sendpdu.checksum = setchecksum(sendpdu);

//始终存win个sendbuf{%win}

sendbuf[(sendpdu.seq / (nsize + 1)) % sendpdu.win] = sendpdu;

bufflag[(sendpdu.seq / (nsize + 1)) % sendpdu.win] = 1;

cout << "保存seq：" << sendpdu.seq << "在[" << (sendpdu.seq / (nsize + 1)) % sendpdu.win << "]中" << endl;

sendret = send(sendpdu);//发送

if (sendret < 0) {

cout << "send error" << GetLastError() << endl;

return;

}

else {

cout << "=======================================================send=======================================================" << endl;

pdu\_to\_str(sendpdu);

}

if (!sign) {

sendpdu.seq = sendpdu.seq + sendpdu.length + 1;//设置下一次将要发送的seq

}

else {

cout << "sendthread\_finished\_will\_return" << endl;

cv.notify\_one();

}

cout << "sendThread\_exit" << endl;

}

return;

}

接收线程：

void threadrecv(uint32\_t& start\_index, uint32\_t& end\_index) {

Sleep(1000);//先进行send

while (1) {

std::unique\_lock<std::mutex> lck(mts);

cout << "===recvThrea\_go\_on===" << endl;

if (first == 0) {//第一次先将发送窗口都发完

cv.notify\_one();

cv.wait(lck);

}

//判断是否超时重发

timeout\_retry(sendbuf[(lastsendbufseq / (nsize + 1)) % sendpdu.win]);

if ((recvpdu.ack - recvpdu.seq == 1 && recvpdu.flags == ACK\_SYN)) {

cout << "处理多包" << endl;

continue;

}//处理建立连接多出来的包

//收到srv的ACK进行检查

if (recvpdu.flags == ACK\_FIN) {

cout << "接收端接收完毕，不必继续等待ACK确认包" << endl;

return;

}

//最后一个包已经发送完成，判断是否得到窗口发送的全部的ACK包

if (sendbuf[(lastsendbufseq / (nsize + 1)) % sendpdu.win].B\_E == 1) {

end\_index = lastsendbufseq;

//&&

//((((lastsendbufseq + sendpdu.length + 1) == recvpdu.ack) && recvpdu.flags == ACK))||flag[(lastsendbufseq / (nsize + 1)) % sendpdu.win]){

int index = ((recvpdu.ack - sendpdu.length - 1) / (nsize + 1)) % sendpdu.win;

if (((lastsendbufseq + sendpdu.length + 1) == recvpdu.ack) && recvpdu.flags == ACK)

flag[index] = 1;//确认ACK

bool finish = true;

for (int seq = start\_index; seq <= lastsendbufseq; seq += nsize + 1) {

if (flag[seq / (nsize + 1) % sendpdu.win] == 0) {

cout << "===========================BEFORE\_ACK\_NOT\_RECV\_YET\_resend\_PDU:=======================" << endl;

send(sendbuf[seq / (nsize + 1) % sendpdu.win]);

pdu\_to\_str(sendbuf[seq / (nsize + 1) % sendpdu.win]);

finish = false;

}

}

if (finish)

{

cout << "最后一个数据包的ACK已经接收，结束传输过程===recv\_thread\_return" << endl;

return;

}

}

//如果接收到窗口下界的ack且ACK，判断连续，更新窗口，并转向发送线程

else if ((recvpdu.ack == start\_index + sendpdu.length + 1) && recvpdu.flags == ACK) {

int index = ((recvpdu.ack - sendpdu.length - 1) / (nsize + 1)) % sendpdu.win;

flag[index] = 1;//确认ACK

cout << "=========RECV\_LOWER\_BOUND============seq=>" << recvpdu.ack - sendpdu.length - 1 << "=>" << index << endl;

check\_ACK\_update(index, start\_index, end\_index, sendpdu.win);

cv.notify\_one();//及时发送滑动窗口内

cv.wait(lck);

continue;

}

//收到窗口上界ACK但还有部分没有确认

else if ((recvpdu.ack == end\_index + sendpdu.length + 1) && recvpdu.flags == ACK) {

int index = ((recvpdu.ack - sendpdu.length - 1) / (nsize + 1)) % sendpdu.win;

flag[index] = 1;

cout << "=========RECV\_HIGHER\_BOUND============seq=>" << recvpdu.ack - sendpdu.length - 1 << "=>" << index << endl;

int start = (start\_index / (nsize + 1)) % sendpdu.win;

int end = (end\_index / (nsize + 1)) % sendpdu.win;

if (start <= end)

for (int i = start; i <= end; i++) {

if (flag[i] == 0) {

cout << "=========before\_ACK\_not\_recv\_yet\_resend\_PDU============seq[" << i << "]=>" << sendbuf[i].seq << "=+>" << flag[i] << endl;

send(sendbuf[i]);

pdu\_to\_str(sendbuf[i]);

}

else {

cout << "ACK\_seq[" << i << "]=>" << sendbuf[i].seq << endl;

}

}

else {

for (int i = start; i < sendpdu.win; i++) {

if (flag[i] == 0) {

cout << "=========before\_ACK\_not\_recv\_yet\_resend\_PDU============seq[" << i << "]=>" << sendbuf[i].seq << "=+>" << flag[i] << endl;

send(sendbuf[i]);

pdu\_to\_str(sendbuf[i]);

}

else {

cout << "ACK\_seq[" << i << "]=>" << sendbuf[i].seq << "=+>" << flag[i] << endl;

}

}

for (int i = 0; i <= end; i++) {

if (flag[i] == 0) {

cout << "=========before\_ACK\_not\_recv\_yet\_resend\_PDU===========seq[" << i << "]=>" << sendbuf[i].seq << "=+>" << flag[i] << endl;

send(sendbuf[i]);

pdu\_to\_str(sendbuf[i]);

}

else {

cout << "ACK\_seq[" << i << "]=>" << sendbuf[i].seq << endl;

}

}

}

}

//接收到窗口范围内乱序ACK，标记确认接收

else if (((recvpdu.ack > start\_index + sendpdu.length + 1)

&& (recvpdu.ack < end\_index + sendpdu.length + 1 || recvpdu.ack == end\_index + sendpdu.length + 1)

&& recvpdu.flags == ACK)) {

int index = ((recvpdu.ack - sendpdu.length - 1) / (nsize + 1)) % sendpdu.win;

if (flag[index] == 1) { continue; }//重复的ACK

flag[index] = 1;

cout << "接收到范围内的乱序ACK:对应seq=>" << recvpdu.ack - sendpdu.length - 1 << "=>" << index << endl;

continue;

}

}

}

//检查连续已经确认的数据包并据此更新发送窗口范围，唤醒发送线程发送

void check\_ACK\_update(int start, uint32\_t& start\_index, uint32\_t& end\_index, int winsize) {

int end;

int e = (end\_index / (nsize + 1)) % sendpdu.win;

//(recvpdu.ack == start\_index + sendpdu.length + 1) && recvpdu.flags == ACK

cout << "bufflag" << start << "=>" << flag[start] <<"in["<<start\_index<<","<<end\_index<<"]"<< endl;

if (flag[start] == 1 && recvpdu.ack == start\_index + sendbuf[start].length + 1) {

if (start <= e) {

for (int i = start; i <= e; i++) {

if (flag[i] == 0) {

cout << "CHECK\_seq+=>" << sendbuf[i].seq << "=>flag:=>" << i << "==" << flag[i] << "WRONG" << endl;

end = i - 1;

break;

}

else {

cout << "CHECK\_seq+=>" << sendbuf[i].seq << "=>flag:=>" << i << "==" << flag[i] << "RIGHT" << endl;

flag[i] = 0;//清空标志

end = i;

}

}

}

else {

bool f = true;

for (int i = start; i < sendpdu.win; i++) {

if (flag[i] == 0) {

end = i - 1;

cout << "CHECK\_seq+=>" << sendbuf[i].seq << "=>flag:=>" << i << "==" << flag[i] << "---+" << "WRONG" << endl;

f = false;

break;

}

else {

cout << "CHECK\_seq+=>" << sendbuf[i].seq << "=>flag:=>" << i << "==" << flag[i] << "---+" << "RIGNT" << endl;

flag[i] = 0;//清空标志

end = i;

}

}

cout << "这个end=" << end << "这个f" << f << endl;

if (f){//要继续判断

for (int i = 0; i <= e; i++) {

if (flag[i] == 0) {

if (i == 0) { end = sendpdu.win - 1; }

else end = i - 1;

cout << "CHECK\_seq+=>" << sendbuf[i].seq << "=>flag:=>" << i << "==" << flag[i] << "---+" << "WRONG" << endl;

break;

}

else {

cout << "CHECK\_seq+=>" << sendbuf[i].seq << "=>flag:=>" << i << "==" << flag[i] << "RIGHT" << endl;

flag[i] = 0;//清空标志

end = i;

}

}

}

}

}

//确认连续小边界，更新滑动窗口

cout << "end=" << end;

int next = sendbuf[end].seq + sendbuf[end].length + 1;

update\_winval(start\_index, end\_index, sendpdu.win, next);

}

**//接收方：循环接收，见报告最开始的部分**

while (1) {

recvret = recv();

if (recvpdu.seq > lastrecvseq) {

lastrecvseq = recvpdu.seq;

}

if (recvret == -1 && GetLastError() == 10060) {

sendret = send();

cout << "==================================================TIME\_OUTresend===================================================" << endl;

pdu\_to\_str(sendpdu);

updateouttime();

cout << "timeout happen,so more timeout" << timeOut << endl;

}

if (recvret > 0) {

cout << "=====================================================recv==================================================" << endl;

lessouttime();

cout << "success recv,so less timeout" << timeOut << endl;

if (recvpdu.flags != ACK || recvpdu.seq == 1) { continue; }//处理建立连接的多余数据包

pdu\_to\_str(recvpdu);

//校验失败，继续接收

if (checkchecksum(recvpdu) == 0) { cout << "check faild" << endl; continue; }

//在接收范围内的数据包:

//1.接收到最下界，判断连续，写入，更新滑动窗口和bufflag标志（顺序接收到最下界，直接写入；后续接收到最下界，去判断缓存的连续）

//2.乱序接收，即收到的不是最下界

//上述结束后，之后如果接收到最后一个包或者窗口上界，检查缓存窗口内的丢包，发送ACK\_LOSS

if (recvpdu.seq >= start\_index && recvpdu.seq <= end\_index && recvpdu.flags == ACK) {

//确认后将接收到的数据写入本地文件

if (recvpdu.seq == start\_index)

{

if (recvpdu.seq <= lastrecvseq) {//说明收到的是丢包且位于最下界

cout << recvpdu.seq << "小于 " << lastrecvseq << "保存seq：" << recvpdu.seq << "在[" << (recvpdu.seq / (nsize + 1)) % recvpdu.win << "]中" << endl;

int index = (recvpdu.seq / (nsize + 1)) % recvpdu.win;

bufflag[index] = 1;

recvbuf[(recvpdu.seq / (nsize + 1)) % recvpdu.win] = recvpdu;

sendpdu.ack = recvpdu.seq + recvpdu.length + 1;//设置发送ack.下一个希望接受的seq

sendpdu.flags = ACK;//设置ACK

sendret = send();//发送ACK包

cout << "======================================================send==================================================" << endl;

pdu\_to\_str(sendpdu);

if (checkrecvbuf\_and\_update\_win(index, outf, start\_index, end\_index))//检查缓冲区是否有下边界的连续包，有就交给上层并更新接收范围

{

break;

}

}

else {

cout << "写入seq=>:" << start\_index << endl;

outf.write(recvpdu.buf, recvpdu.length);

sendpdu.ack = recvpdu.seq + recvpdu.length + 1;//设置发送ack.下一个希望接受的seq

next\_want\_seq = sendpdu.ack;//设置下一个期望的recv的seq

sendpdu.flags = ACK;//设置ACK

sendret = send();//发送ACK包

cout << "======================================================send==================================================" << endl;

pdu\_to\_str(sendpdu);

if (recvpdu.B\_E == 1) {

cout << "RECV finished!!!" << endl; recvpdu.length = 0;//接收完成

break;

}

update\_winval(start\_index, end\_index, sendpdu.win, sendpdu.ack);

}

}

//乱序接收

else {

cout << "=================================================OutOfOrderbuf=====================================================" << endl;

//存下recvpdu

int index = (recvpdu.seq / (nsize + 1)) % recvpdu.win;

recvbuf[(recvpdu.seq / (nsize + 1)) % recvpdu.win] = recvpdu;

bufflag[index] = 1;//标志

cout << "保存seq：" << recvpdu.seq << "在[" << index << "]中=>+" << bufflag[index] << endl;

sendpdu.ack = recvpdu.seq + recvpdu.length + 1;//设置发送ack.下一个希望接受的seq

sendpdu.flags = ACK;//设置ACK

sendret = send();//发送ACK包

cout << "======================================================send==================================================" << endl;

pdu\_to\_str(sendpdu);

}

}

//在接受范围窗口之外的ACK包

//小于窗口范围

else if (recvpdu.seq < start\_index && recvpdu.flags == ACK) {

//确认包丢失重传

sendpdu.flags = ACK;

sendpdu.ack = recvpdu.seq + recvpdu.length + 1;//设置发送ack.下一个希望接受的seq

sendret = send();//发送ACK包

cout << "==================================================ACK\_LOSS====send==================================================ack=>"<<sendpdu.ack << endl;

pdu\_to\_str(sendpdu);

}

//接收完成

}

}

//检查接收缓冲区的连续文件并写入文件，更新接收缓冲区范围

int checkrecvbuf\_and\_update\_win(int start,ofstream &fout,uint32\_t&start\_index,uint32\_t&end\_index) {

int end = -2;

int e = (end\_index / (nsize + 1)) % sendpdu.win;

cout << "收到后来的下界，故开始检查连续并写入in[" << start << "," << e << "]" << endl;

cout << "bufflag" << start << "=>" << bufflag[start] << endl;

if (bufflag[start] == 1 && recvbuf[start].seq == start\_index) {

if (start <= e) {

for (int i = start; i <= e; i++) {

if (bufflag[i] == 0) {

end = i - 1;

break;

}

else {

end = i;

cout << "写入seq=>:" << recvbuf[i].seq << endl;

fout.write(recvbuf[i].buf, recvbuf[i].length);

bufflag[i] = 0;//清空标志

if (recvbuf[i].B\_E == 1) {

cout << "RECV finished!!!" << endl; recvpdu.length = 0;//接收完成

return 1;

}

}

}

}

else {

bool f = true;

for (int i = start; i < sendpdu.win; i++) {

if (bufflag[i] == 0) {

end = i - 1;

f = false;

break;

}

else {

end = i;

cout << "写入seq=>:" << recvbuf[i].seq << endl;

fout.write(recvbuf[i].buf, recvbuf[i].length);

bufflag[i] = 0;//清空标志

if (recvbuf[i].B\_E == 1) {

cout << "RECV finished!!!" << endl; recvpdu.length = 0;//接收完成

return 1;

}

}

}

if (f){//需要继续判断

for (int i = 0; i <= e; i++) {

if (bufflag[i] == 0) {

end = i - 1;

if (i == 0)end = sendpdu.win - 1;

break;

}

else {

end = i;

cout << "写入seq=>:" << recvbuf[i].seq << endl;

fout.write(recvbuf[i].buf, recvbuf[i].length);

bufflag[i] = 0;//清空标志

if (recvbuf[i].B\_E == 1) {

cout << "RECV finished!!!" << endl; recvpdu.length = 0;//接收完成

return 1;

}

}

}

}

}

cout << "end" << end<<" => "<< recvbuf[end].seq<<"=>"<< recvbuf[end].length + 1 << endl;

update\_winval(start\_index, end\_index, sendpdu.win, recvbuf[end].seq + recvbuf[end].length + 1);

}//连续范围[start,end]并写入

return 0;

}

## 运行截图以及分析

##### 接收端：输入路由端口，绑定端口，和接收窗口的大小建立连接后输入保存的文件名

##### 

##### 

##### 发送端：输入路由端口，绑定端口，发送窗口大小，建立连接后输入传输文件路径

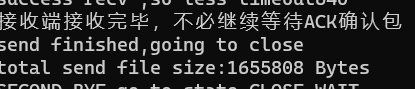
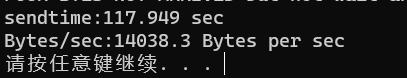
##### 

##### 

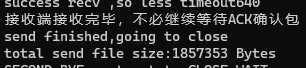
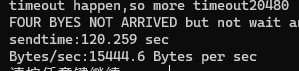
1.设置时延30ms,丢包率30%,滑动窗口数32，传输结果（文件大小+传输时间

）如下：均能传输成功，且文件大小与原大小一致

传输helloworld.txt：

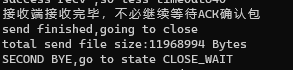
传输1.jpg：

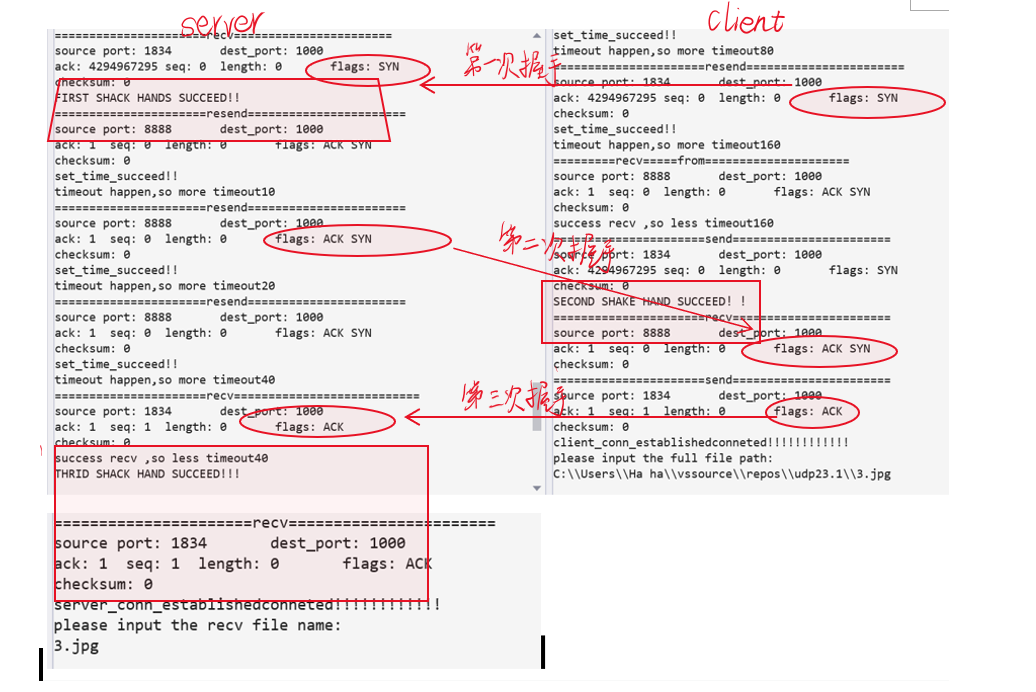
传输2.jpg：

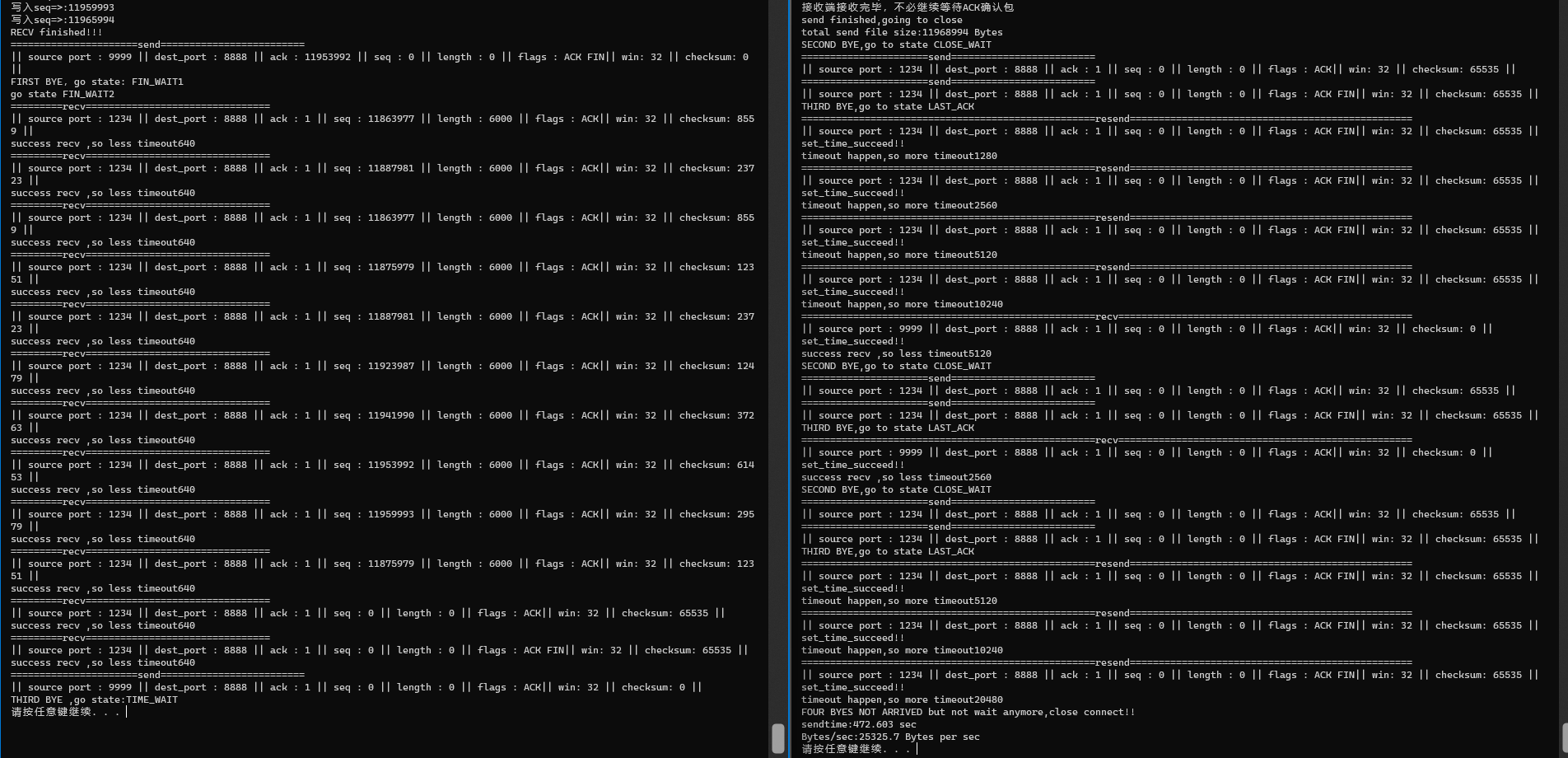
传输3.jpg：



三次握手如下：



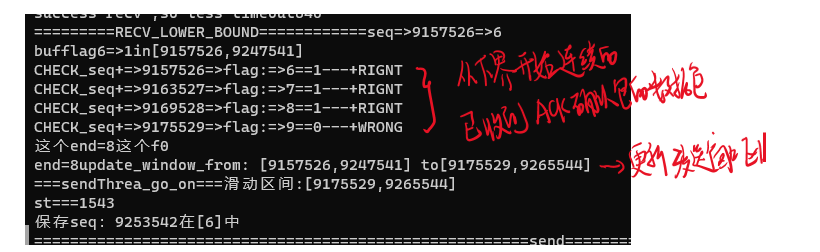
四次挥手如下：接收端先关闭连接



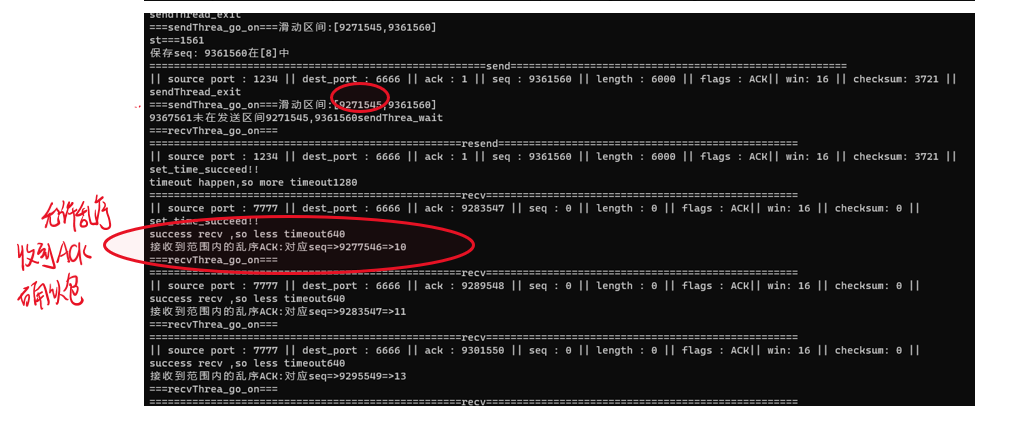
以下是以30%丢包率30ms延时传输3.jpg为例

发送方的选择确认如下：

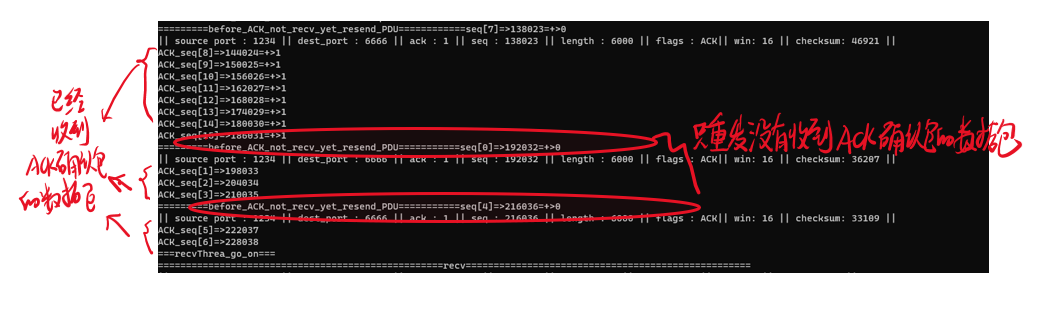
收到下界，检查从窗口下界开始连续数据包，更新窗口：



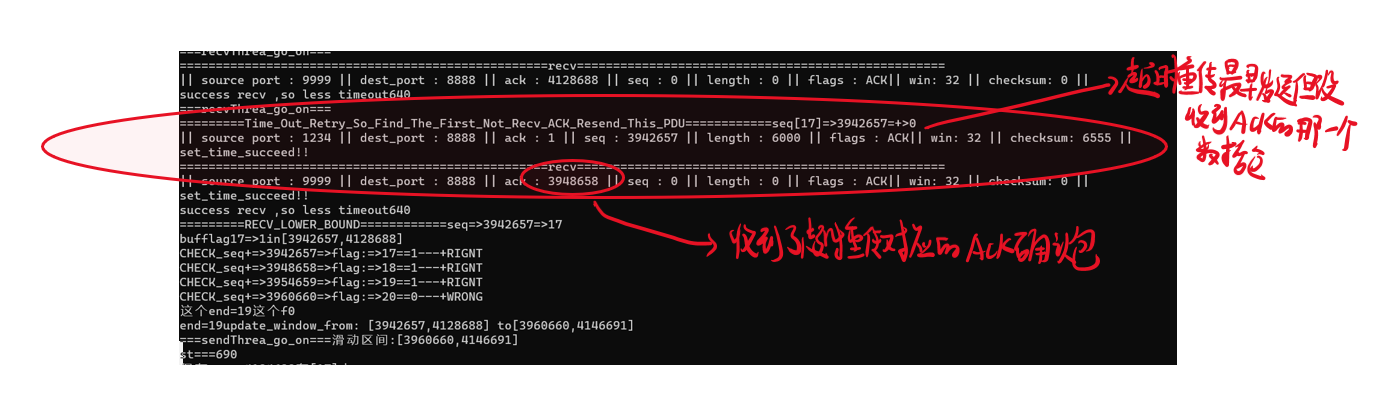
允许在发送窗口范围内乱序接收ACK确认包



只重发没有接收到ACK确认包的数据包

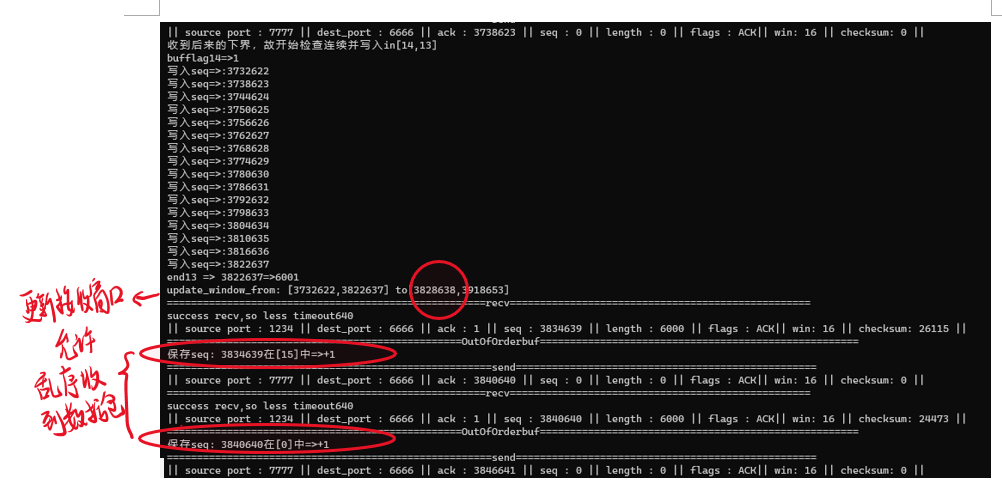


超时重传窗口会重传发送窗口内最早发送但没有收到确认的包

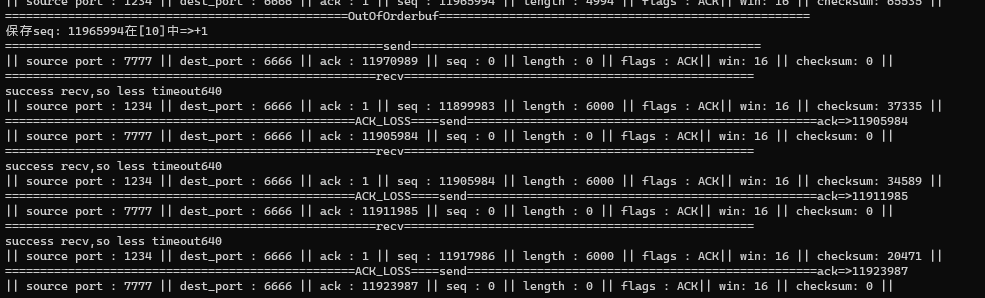


接收端：

检查缓存中从下界连续的数据包，写入，并更新接收窗口范围，将乱序到达的窗口范围内的数据包缓存



收到小于窗口下界的数据包，说明ACK确认包丢失，重传ACK确认包



## 实验中的问题

之前对接收方也做了=>收到大于等于接收窗口上界时检查丢包的步骤，并发送对应的ACK\_LOSS包，发送方对ACK\_LOSS和没有接收到ACK的数据包都进行了重发，但是这两者会有重复发送的部分，所以删掉了接收方收到大于等于接收窗口上界检查丢包的部分

对于三次握手和四次挥手的最后一次ACK确认包的到达并不确认，如果超过设定的最大超时时间，就会直接建立连接或者关闭连接，但不影响数据包传输的正确性

在之前实验中，由于都是发送方先关闭，所以没有进行接收方先关闭的处理；在此次实验中，接收方接收完后可以先发送关闭请求，发送方接收到后不再继续等待ACK确认包，也开始关闭，故双方都增加了先(后)关闭的函数

检查之前对于超时重传没有设计准确，只会重传发送线程中最近发送的那个数据包（即窗口上界），相当于只依赖于收到窗口上界的ACK就会重发窗口内没有收到ACK的数据包的重传，故将超时重传修改为重传发送窗口中最早发送但还没接收到ACK确认包的那一个。