# 实验3：基于UDP服务设计可靠传输协议并编程实现（3-2）

## 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：**

超时的话就重发最近一次发送的数据包，判断接收到的确认包的确认号只需要位于[start\_index+senpdu.length+1, end\_index+senpdu.length+1]即可，

如果ACK确认包正确且ack大于上一次接收到的正确ack就更新滑动窗口覆盖的字节序号，唤醒发送线程发送滑动窗口内的剩余数据包；

当收到ACK\_LOSS丢失包时，重发从丢失包到之前发送线程最后一次发送的包；

当收到滑动窗口内的未发送的ACK确认包（导致这个的原因是因为在接收线程中更新滑动窗口有两处{

一处是接收正确范围内的ACK，就会唤醒发送线程进行发送，接收线程进入阻塞;

一处是在接收到ACK\_LOSS时，进行再次发送从丢包处到发送线程最近一次发送的包处之间的多个数据包并接收到正确范围内的ACK后，此处更新滑动窗口后不会唤醒发送线程，而是继续接收}

这样设置的目的是为了让发送和接收交替频繁且随机一点，因为考虑到丢包的话更新滑动窗口就立刻发送可能收到的还是ACK\_LOSS，增加网络拥塞），就唤醒发送线程发送滑动窗口内的剩余数据包

#### 接收方：

循环接收：

设置累计确认wait个包（设置为发送端的窗口大小）

对于正确顺序的数据包，设置sendpdu为对应ACK确认包，wait--，当累计收到wait个包后就发送一次ACK，并将wait恢复为原大小（发送端的发送窗口大小）；将接收数据写入指定文件，当收到具有结束标志的数据包，写入文件后结束循环

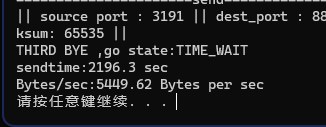
对于不正确的数据包：

1. 对于序列号小于next\_want\_seq（期望接受到的序列号）且最近一次接收方更新sendpdu为ACK\_LOSS的数据包（此类数据包是重复的数据包），直接丢弃
2. 对于序列号小于next\_want\_seq（期望接受到的序列号）的数据包且最近一次接收方更新sendpdu不是ACK\_LOSS的数据包{也就是更新的sendpdu为ACK确认包} (这可能由于发送方的超时重传或者是由于ACK确认包丢失)，就会重发(或发送)已经设置好的ACK确认包{主要目的是为了使发送ACK确认包动态一点，因为对于发送方发送较快的情况{例如：一下发送多个或者由于丢包连续发送多个}，可以在短时间内接收多个数据包而只返回最后一个数据包的ACK确认包，可以减少包的数量}

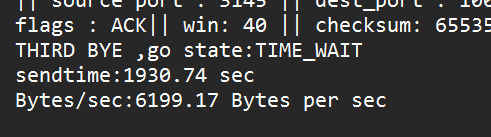
Ps：我设置30%丢包率，30ms延时，40窗口大小，发送3.jpg，分别使用一次一发ACK的程序和代码中设计的累计确认的程序，使用所给的路由器，测试结果如下：

{累计n个确认+超时重传约束看起来更好一点：我觉得是因为滑动窗口的连续发送会让接收端短时间接收多个数据包，如果每个都返回ACK，降低了传输效率，特别是在窗口数很多的情况下（考虑到所给的路由丢包好像是规律丢包，可能据此测效率意义不大，加上我本意想让滑动窗口的发送动态一点<设置更新滑动窗口有时会导致发送，有时不会导致发送，就使得连续发多个包的情况更多，所以累积多个+超时重传的约束的结果会更好一点>）}

收到正确seq就发送ACK：



累计n个确认+超时重传约束：



2.对于序列号大于next\_want\_seq的数据包，说明next\_want\_seq的数据包丢失，向发送方发送ACK\_LOSS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

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

### int lastrecvack=0;//记录最大的正确ack

//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);

### 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

#### 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)

##### 累计确认（ACK确认包可以丢失，只要ACK确认包位于滑动窗口范围即可，对于发送端发送较快的情况，可以对几个连续正确数据包只返回一个ACK确认包）

##### 对接收方：

##### 设置累计确认数wait为发送窗口大小，初始化为默认值

设置next\_want\_seq=0//希望接收到的序列号（为上次接收到的序列号+接收数据大小+1）

对于每次有效的recv()，

1. 检验16位校验和，如果相加不为0xffff,就丢弃该包，继续接收
2. 判断接收到的序列号是否等于next\_want\_seq,如果相等就将接收数据写入文件；

并设置要发送的确认包{ACK+(ack=recvpdu.seq+recvpdu.length+1)}；

并更新next\_want\_seq=ack;

累计确认数wait--；如果wait为0，就发送ACK确认包，并恢复wait为发送窗口数

如果收到的包中有B\_E=1（文件发送结束标志），就发送ACK确认包，跳出循环接收

继续循环接收

1. 判断接收到的序列号小于期望序列号：

对于重复的数据包，就丢弃，并wait--{因为可能是由于发送方超时重传而造成的，所以减少累计确认数}

对于ACK确认包丢失或者由于等待累计确认个数造成的超时重传现象的情况，就重发ACK确认包

1. 接收到的序列号大于期望序列号：

发生丢包，发送ACK\_LOSS代表丢包

继续循环接收

##### 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

对接收到的ACK包中的ack进行判断：

1. 若在限制时间内接收到的ack位于滑动窗口大小对应的期望收到的ack范围之内,说明收到正确的确认包，并及时更新滑动窗口覆盖字节序列的范围 ；超时就重发最近一次发送线程发送的数据包
2. 若收到滑动窗口外的ACK包，判断发送缓存中是否有对应的数据包，没有说明滑动窗口中存在未发送的数据包，唤醒发送线程，接收线程阻塞，将剩余数据包发送后，唤醒接收线程，发送线程阻塞。
3. 若收到滑动窗口范围内的ACK\_LOSS包，就从发送缓存中取出丢失的包以及在此之后已经发送过的包，然后进行接收，若接收到正确范围内的ack包就更新滑动窗口覆盖字节序列的范围（此处不立即唤醒发送线程是觉得此时可能丢包现象造成较多数据包的重复发送），若未正确接收，重复1-2

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

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.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() {

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;

timeout\_retry(sendpdu);

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

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

Sleep(10000);

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);}

//接收方

bool close\_conn() {

//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

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;

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**

接收到正确范围内的ACK就更新滑动窗口，可能会唤醒发送线程发送；

如果接受到位于滑动窗口但还未发送的ACK,就唤醒发送线程发送完剩下可发送的数据包再切换为接收线程；

接收到ACK\_LOSS，如果是已发送已缓存的数据包，就从该丢包处起，至发送线程最近一次发送的数据包止，再次发送此区间内的数据包

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

Sleep(1);//先进行send

while (1) {

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

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

//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)) {

continue;

}//处理建立连接多出来的包

//收到srv的ACK进行检查

//最后一个包已经发送完成，且得到最后的ACK包

if (sendbuf[(lastsendbufseq / (nsize + 1)) % sendpdu.win].B\_E == 1 && ((lastsendbufseq + sendpdu.length + 1) == recvpdu.ack) && recvpdu.flags == ACK) {

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;

}

//如果接收到正确范围内的ack且ACK，且不是重复的ack就更新滑动窗口，转向发送线程

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

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

&& recvpdu.flags == ACK) {

if (lastrecvack < recvpdu.ack) {

lastrecvack = recvpdu.ack;

update\_winval(start\_index, end\_index, sendpdu.win, recvpdu.ack);//更新滑动窗口

if (!sign) {

cv.notify\_one();//可能此时网络通畅，就及时发送

cv.wait(lck);

}

continue;

}

else {

cout << "丢弃晚到的较小ack:" << recvpdu.ack << "<=" << lastrecvack << endl;//将晚到的较小的ack丢弃

continue;

}

}

//接收到ack\_loss的包或者滑动窗口还有可以发送的数据包

else {

while (!

((recvpdu.ack == start\_index + sendpdu.length + 1 || recvpdu.ack > start\_index + sendpdu.length + 1)

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

&& recvpdu.flags == ACK)) {

cout << "ack check failed,may loss ,will resend" << endl;

//在recv前滑动窗口存在未发送的数据包

if (bufflag[((recvpdu.ack) / (nsize + 1)) % sendpdu.win] == 0 || (sendbuf[((recvpdu.ack) / (nsize + 1)) % sendpdu.win].seq != recvpdu.ack)) {

break;

}

//在此seq后写入的send无效，记录send的最后一次，在此区间的数据包再发送一遍

int shouldsendindex = ((recvpdu.ack) / (nsize + 1)) % sendpdu.win;

cout << "ACK\_LOSS,so重发错误号==>" << recvpdu.ack << "============" << ((recvpdu.ack) / (nsize + 1)) % sendpdu.win << "======" << lastsendbufseq << "====" << (lastsendbufseq / (nsize + 1)) % sendpdu.win << endl;

int lastindex = (lastsendbufseq / (nsize + 1)) % sendpdu.win;

if (!(shouldsendindex > lastindex))

for (int i = shouldsendindex; i < lastindex + 1; i++) {

sendret = send(sendbuf[i]);

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

}

else {

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

cout << "i=>" << i << endl;

sendret = send(sendbuf[i]);

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

}

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

cout << "i=>" << i << endl;

sendret = send(sendbuf[i]);

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

}

}

//超时重发最近发送的数据包

timeout\_retry(sendbuf[shouldsendindex]);

}

//可能滑动窗口还没全部send完，就转向send线程

if (sendbuf[((recvpdu.ack) / (nsize + 1)) % (sendpdu.win)].seq != recvpdu.ack || bufflag[((recvpdu.ack) / (nsize + 1)) % sendpdu.win] == 0) {

cout << "滑动窗口中存在未发送完的数据包:" << recvpdu.ack << "=>" << sendbuf[((recvpdu.ack) / (nsize + 1)) % (sendpdu.win)].seq << " in [" << start\_index << "," << end\_index << "]" << endl;

if (!sign) {

cv.notify\_one();

cv.wait(lck);

}

continue;

}

//更新滑动窗口，继续接收

else {

if (lastrecvack < recvpdu.ack) {

lastrecvack = recvpdu.ack;

update\_winval(start\_index, end\_index, sendpdu.win, recvpdu.ack);//更新滑动窗口

continue;

}

else {

cout << "丢弃较小ACK" << recvpdu.ack << "<=" << lastrecvack << endl;

continue;

}

}

cout << "recvThread\_exit" << endl;

continue;

}

//发送的判断条件只有是否在滑动窗口区间，在此区间就不停发送

//接收和send两个线程，但是有共享的数据区间判断和线程阻塞切换

return;

}

}

**//接收方：循环接收**

累计确认数为发送窗口数，在此期间如果发生ACK确认包丢失或者累计确认造成的超时重传的现象，就重传ACK确认包，如果接收到已经接收过的数据包，说明可能是由于发送方超时重传导致的，就将累计数-1，发送过ACK确认包后再恢复累计确认数=>{主要目的是为了想变动累计确认数}

while (1) {

recvret = recv();

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; }

//确认后将接收到的数据写入本地文件

if (recvpdu.seq == next\_want\_seq)

{

wait--;//累计确认

cout << "写入seq=>:" << next\_want\_seq<<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

if (wait == 0) {//累计确认win个包后发送ACK包

sendret = send();//发送ACK包

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

pdu\_to\_str(sendpdu);

wait = recvpdu.win;

}

//最后一个确认包发送

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

cout << "RECV finished!!!" << endl; recvpdu.length = 0;

sendret = send();//发送ACK包

cout << "======================send\_LAST\_ACK=========================" << endl;

pdu\_to\_str(sendpdu);

break;

}

continue;

}

else if(recvpdu.seq > next\_want\_seq){

//发生丢包

cout << "recv seq=>" << recvpdu.seq << ";but EXPECT recvseq=>" << next\_want\_seq;

//接收的序列号不正确就一直重发ACK\_LOSS<期望得到的seq号>

sendpdu.ack =next\_want\_seq;//设置发送ack.下一个希望接受的seq

sendpdu.flags = ACK\_LOSS;//设置ACK

sendret = send();//发送ACK包

cout << "======================send\_ACK\_LOSS=========================" << endl;

pdu\_to\_str(sendpdu);

continue;

}

else {

//是重复的数据包

if (sendpdu.flags == ACK\_LOSS) {

cout << "duplicated PDU,discard" << endl;

wait--;//缩小累计确认的个数

continue;

}

//可能ACK确认包丢失

sendret = send();//重发ACK包

cout << "======================resend\_ACK========May be ACK loss=================" << endl;

pdu\_to\_str(sendpdu);

continue;

}

//接收完成

}

}

## 运行截图以及分析

##### 接收端：输入路由端口，绑定端口，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，发送的滑动窗口大小，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：



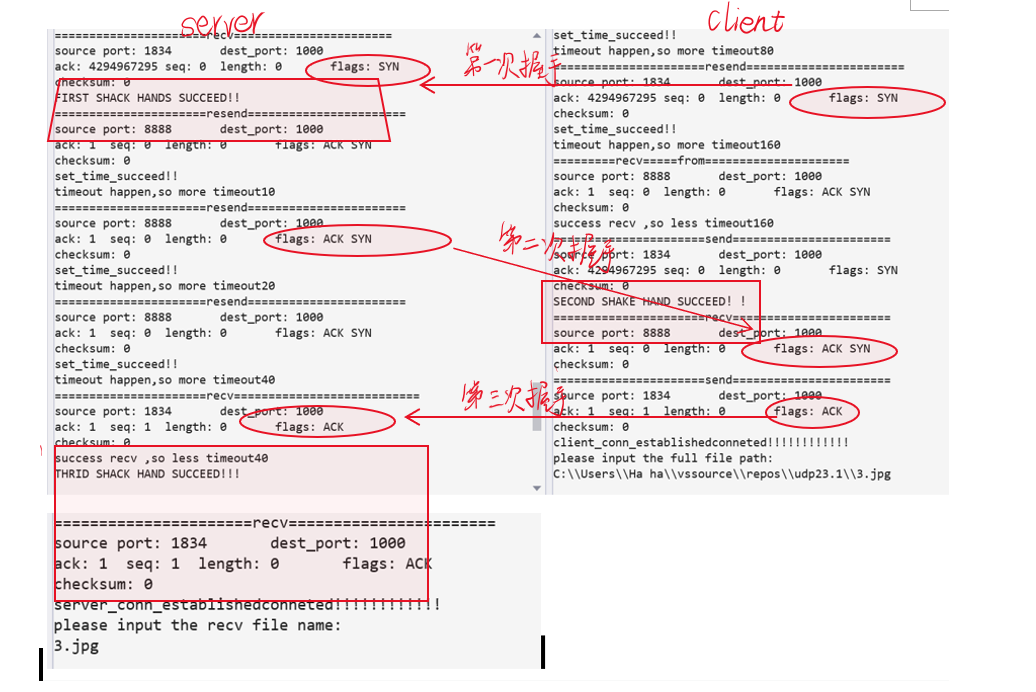
传输2.jpg：



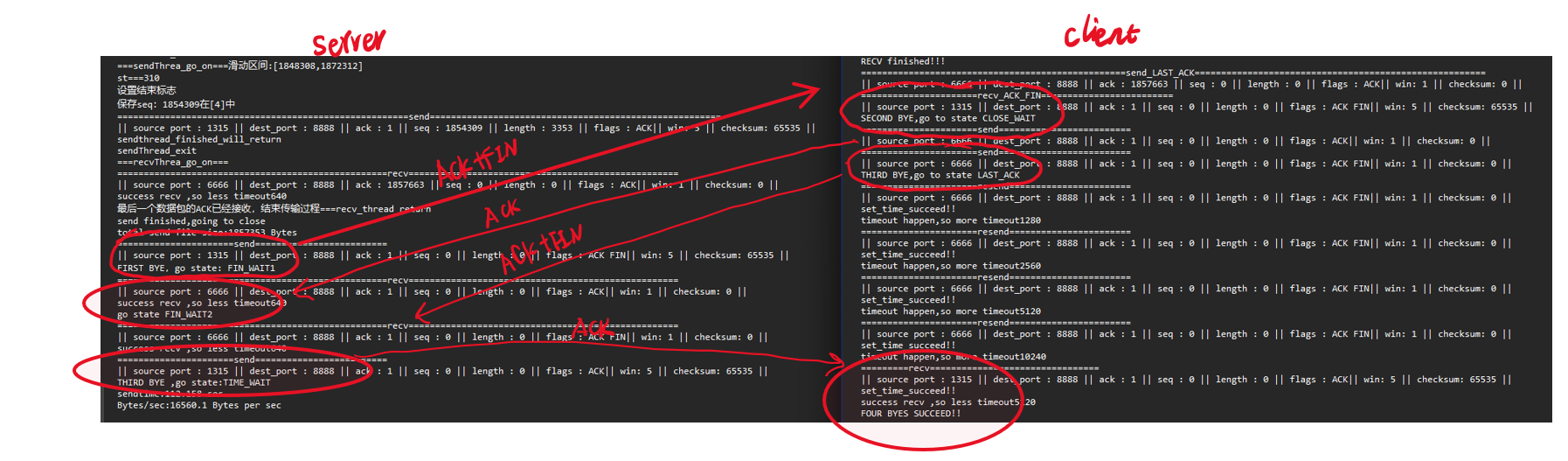
传输3.jpg：

三次握手如下：

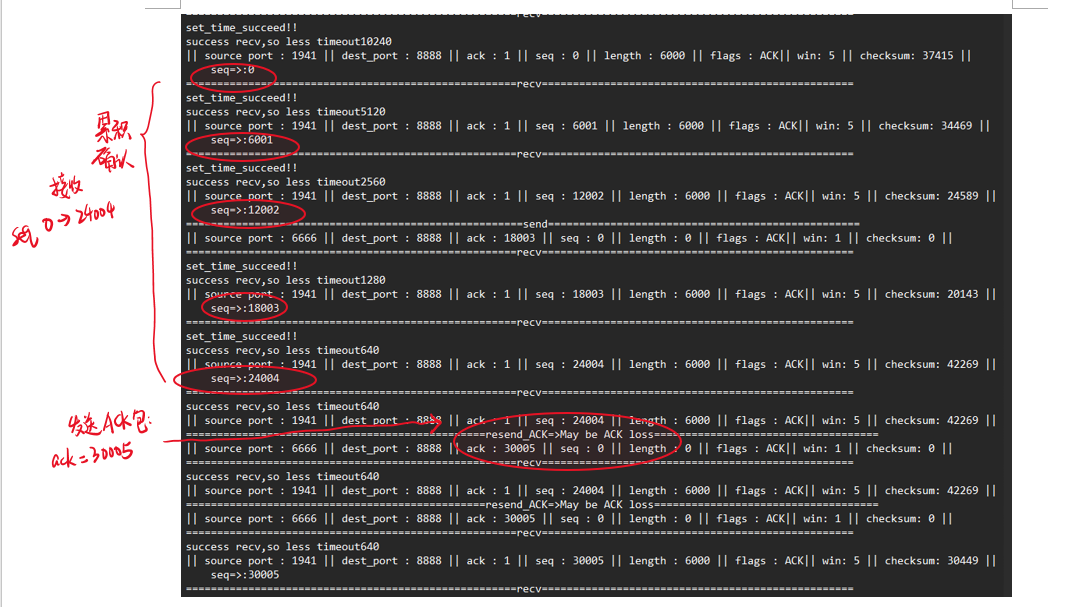


四次挥手（也体现了超时重传的过程）如下



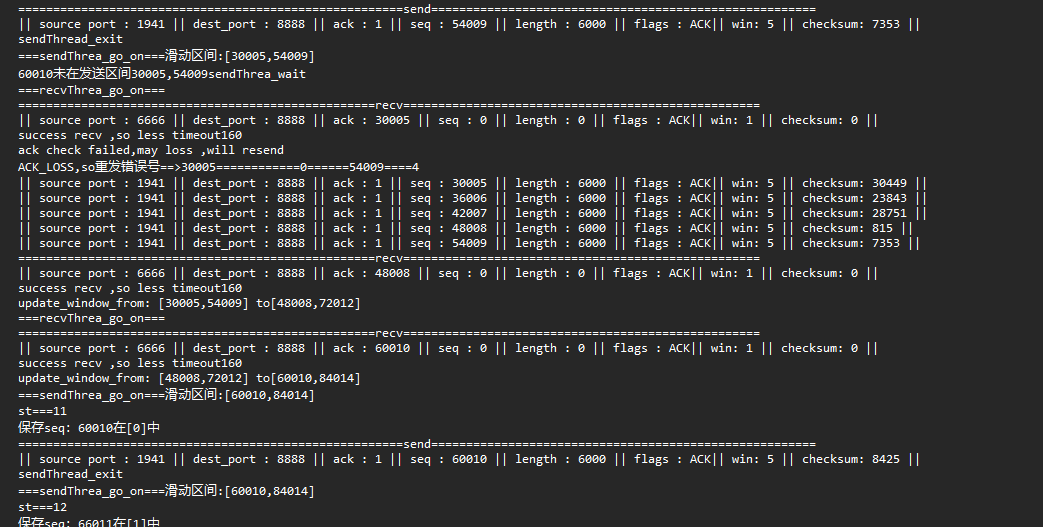
以下是以10%丢包率10ms延时传输1.jpg为例（为了使累计确认更明显，所以丢包率设置少）

接收端的累计确认如下：

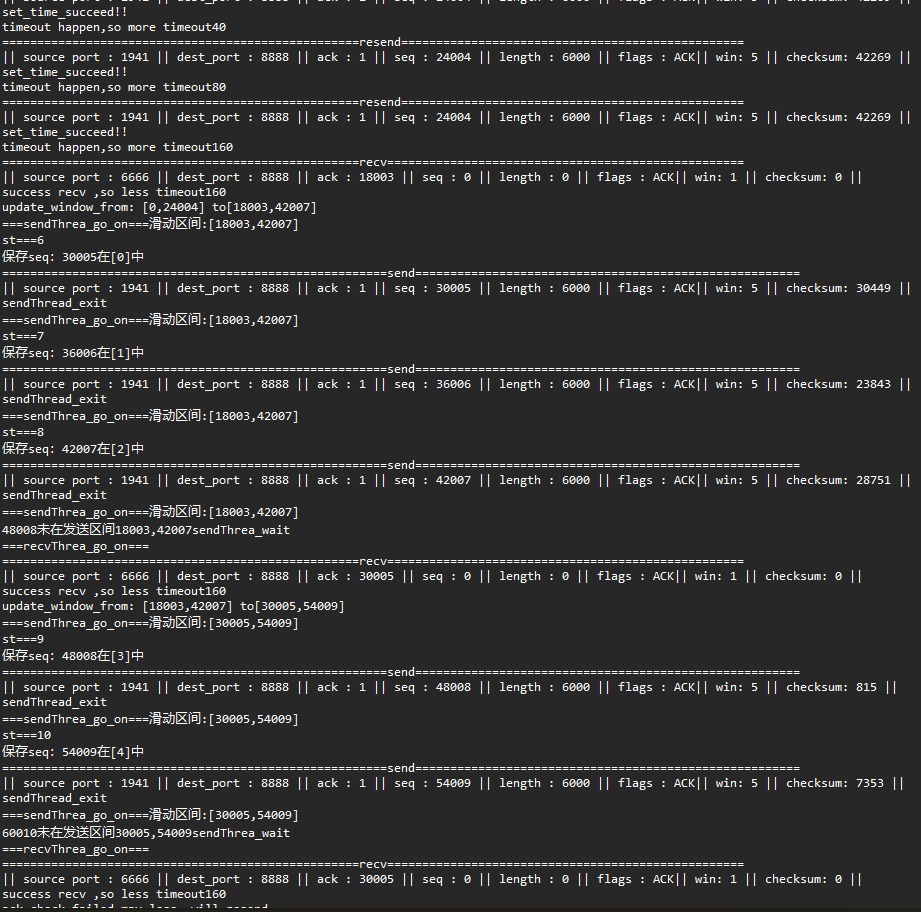


发送端的丢包重传GoBackN和滑动窗口如下：

**传输过程中丢包现象如下：会发送从丢包处到发送线程的最后一次发送的包的一串包：**



**接收到累计确认的ACK后，更新滑动区间并切换到发送线程发送剩余的数据包，如下：**



## 实验中的问题

## 对于发送端的接收和发送交替进行，之前使用锁和Sleep(释放cpu资源)，但是观察传输日志有时候达不到线程切换的目的，所以使用锁+notify\_wait实现了正确的线程切换

对于滑动窗口的丢包问题，每次丢包都导致了接收端将后序收到的包都返回ACK\_LOSS，从而导致了从丢包处开始的一串数据包的重复发送，造成了资源的浪费，容易使发送的数据包越来越多{认为更适合于丢包率高的情况}

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