计算机网络实验三

基于**UDP**服务设计可靠传输协议并编程实现

实验3-2：滑动窗口GBN

姓名：刘伟

学号：**2013029**

专业：物联网工程

一、实验要求

本实验建立在实验3-1的基础上，故一些功能实现未在本报告中加以解释。

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。将 停等机制 改成基于 滑动窗口的流量控制机制 ，采用 固定窗口 大小，支持 累积确认 ，完成给定测试文件的传输。

二、实验协议设计&功能实现

1. 报文格式：

报文设计建立在实验3-1的基础上，在本次实验中使用到了3-1中未使用的window窗口部分，实现了滑动窗口的流量控制机制。

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
datasize数据长度															
			END	OVER	FIN	ACK	SYN	window							
seq序列号								ack确认号							
checksum校验和															

- 24-31位，标记为window窗口，标定了滑动窗口的大小。在实验设计中：发送方在window中填写滑动窗口的空余大小

其余见3-1实验设计

2. 建立连接&断开连接方式：仿照**TCP**协议

该部分与3-1一致，可见3-1实验设计。

3. 差错检验：

该部分与3-1一致，可见3-1实验设计。

4. 双向传输：

该部分与3-1一致，可见3-1实验设计。

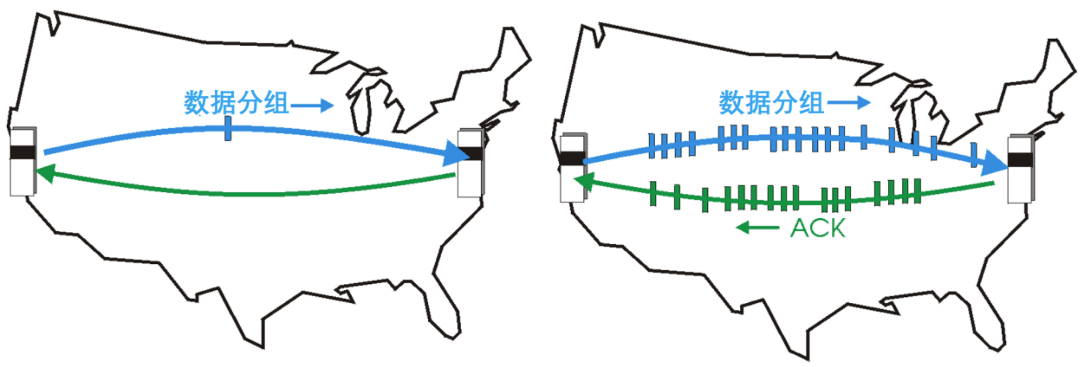
5. 流水线协议

考虑到rdt3.0停等协议的性能问题，未能够充分的提高链路利用率。

在本实验3-2部分采用流水线协议进行性能优化：在确认未返回之前允许发送多个分组。

■ 流水线协议

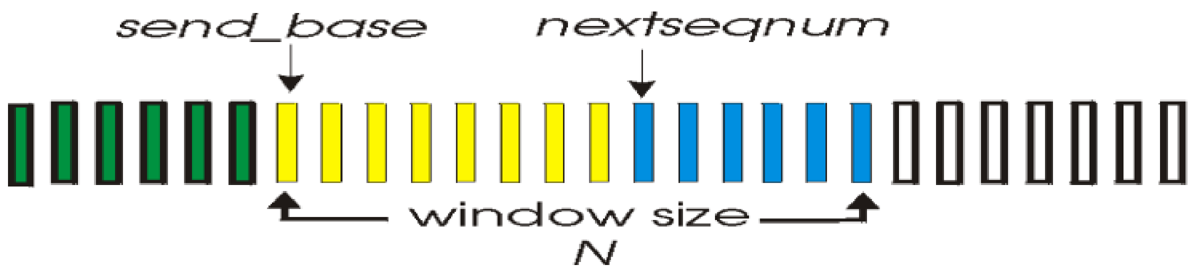
- 流水线机制：在确认未返回之前允许发送多个分组
  - 0和1两个序列号还够用吗？
  - 发送端和接收端缓冲区需要增加吗？



6. 基于滑动窗口的流量控制：

在实验3-2部分，流量控制采用的是`基于滑动窗口的流量控制机制`。

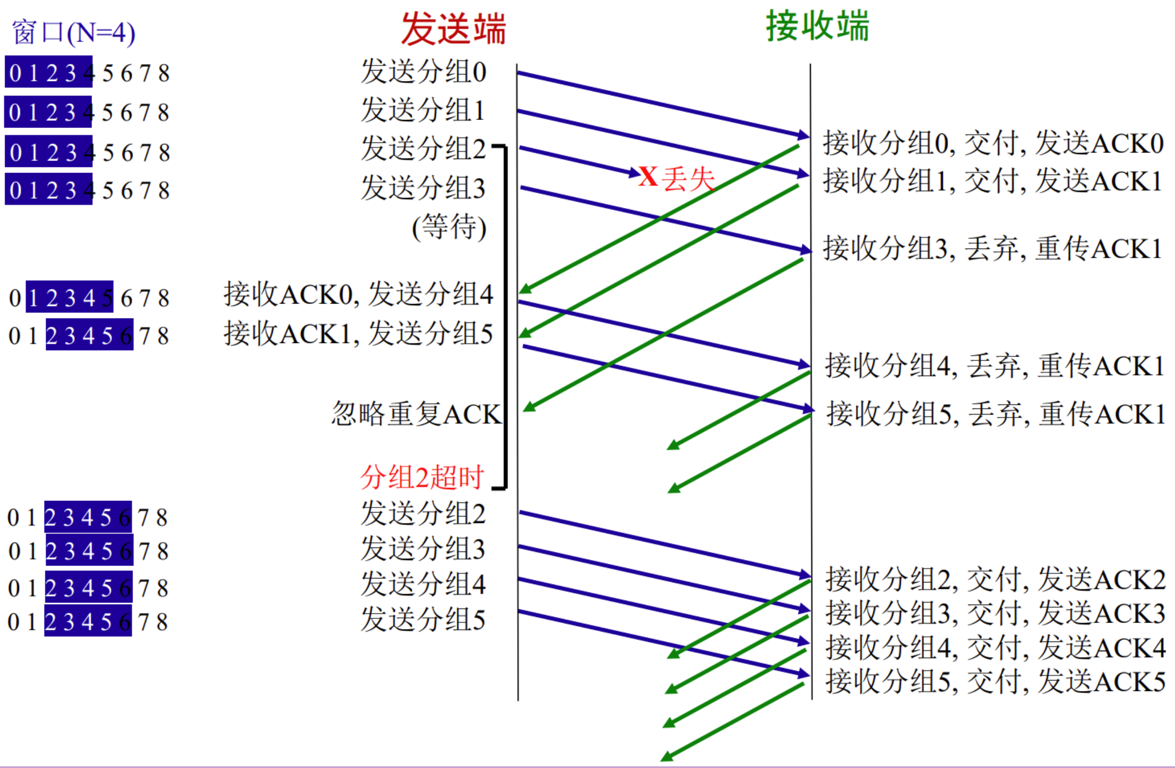
滑动窗口：发送方和接收方各有一个缓存数组，发送方存放着：已发送且成功确认包序号、已发送未确认包序号，未发送包序号。接收方存放着：已接受包序号、正在接收包序号、未接收包序号。每个数组有个两个扫描指针，开头和结尾，一起向后扫描，两者形成一个窗口。假设两端的滑动窗口大小为N，服务器端可以一次性发送N个报文段。以实现流水线机制，目的为提高传输的性能。



窗口：允许使用的序列号范围，窗口尺寸为N：最多有N个等待确认的消息。
滑动窗口：随着协议的运行，窗口在序列号空间内向前滑动。
窗口分为左边界、发送边界和右边界，窗口大小固定。窗口左边界左侧为已经发送并得到确认的数据，左边界到发送边界的数据为已发送但未得到确认的数据，发送边界到右边界为等待发送的数据，右边界右侧为不可发送的数据。

GBN：发送方允许发出N个未得到确认的分组（每次发送的时候都会发送滑动窗口剩余大小个数据包）。发送方采用累计确认，只确认连续正确接受分组的最大序列号。如果在规定时间段内仍没有收到正确的应答信号，那么便启动重传机制，重传所有未被确认的分组。接收方确认按序正确接收的最高序号分组。在GBN的算法下，会产生重复的ACK，需要保存希望接受的分组序号。接收端在面对失序分组的数据包时，不会缓存该数据包（直接丢弃），并重发最高正确序号所对应的ACK，告知发送端。

GBN交互示例

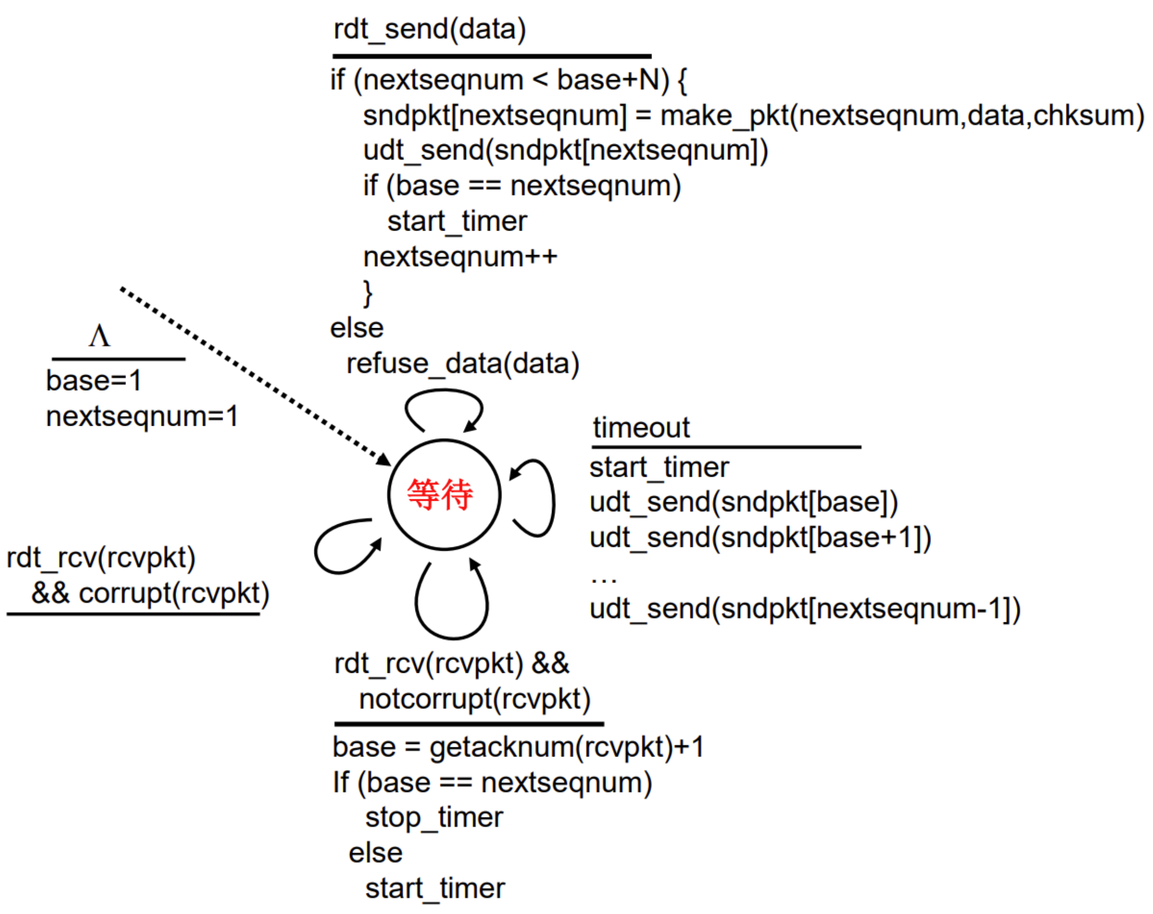


窗口(N=4)
发送端
接收端
0 1 2 3 4 5 6 7 8
发送分组0
发送分组1
发送分组2
发送分组3
(等待)
X 丢失
接收分组0, 交付, 发送ACK0
接收分组1, 交付, 发送ACK1
接收分组3, 丢弃, 重传ACK1
接收ACK0, 发送分组4
接收ACK1, 发送分组5
接收分组4, 丢弃, 重传ACK1
接收分组5, 丢弃, 重传ACK1
忽略重复ACK
分组2超时
发送分组2
发送分组3
发送分组4
发送分组5
接收分组2, 交付, 发送ACK2
接收分组3, 交付, 发送ACK3
接收分组4, 交付, 发送ACK4
接收分组5, 交付, 发送ACK5

7. GBN数据传输

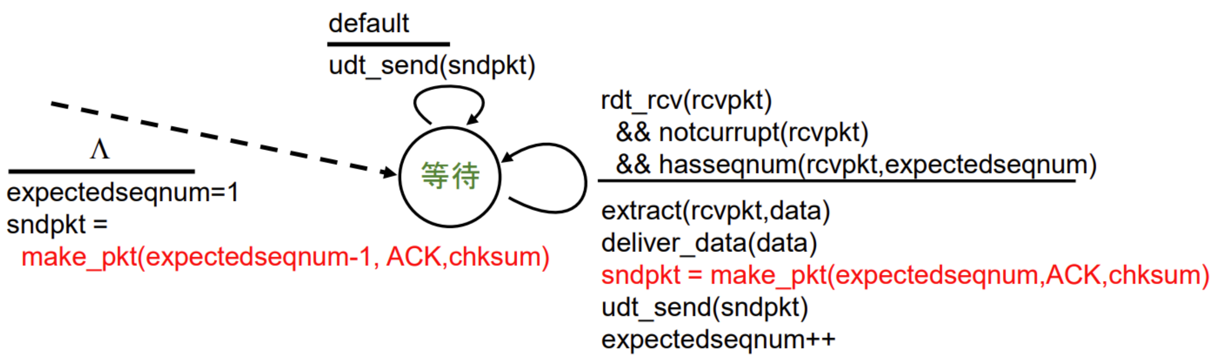
发送方在传输文件数据的时候，会根据单次发送所能容纳的大小将数据进行拆分，分批发送。GBN对于数据的传输算法选取的是滑动窗口的方式。在传输时，无需像停等机制一样非得接受到上一个发送数据包期待的回传响应后才能开始发送下一个数据包，发送方可以继续发送数据包（直到窗口被用完了）。接收端接收到了一个数据包，先要进行校验，如果检查无误，则向发送方返回该序列号的相对应的ACK号。如果数据错误，那么接收方未收到正确按序的数据包，则会回传重发ACK（按序接收的最高序号分组）。发送方在收到重复序列号会采取累计确认的方式。如若发送方在一定的时间段内，都未收到某一数据包的ACK确认响应。那么便会采取超时重传的机制。

- 发送方FSM：



rdt_send(data)
if (nextseqnum < base+N) {
sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
udt_send(sndpkt[nextseqnum])
if (base == nextseqnum)
start_timer
nextseqnum++
}
else
refuse_data(data)
Λ
base=1
nextseqnum=1
等待
timeout
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
stop_timer
else
start_timer

- 接收方FSM：



8. 累计确认&确认重传：

`累计确认` 机制（累计确认指的是当接收到 `n` 号报文段的 `ACK` 报文，表示 `0 - n` 号报文段都已经被成功接收），所以当收到一个 `ACK` 报文时，窗口将向前移动，直到 `base` 的值等于这个 `ACK` 报文所确认的下一个报文的序号为止。而此时，又将有更大的序号进入到窗口中，窗口中将产生空闲区间。

故在GBN实验设计中发送方可能会收到重复的ACK消息，此时发送方需要忽视掉这些重复的ACK。

`累计确认&确认重传` 机制（如果接收到的数据包校验和计算有误或者相关标识出错，那么就需重传缓冲区未被确认的的数据）。

```
if ((compute_sum((WORD*)&packet, sizeof(packet)) != 0)) {  //数据校验出错
    continue;
}
//收到正确响应 滑动窗口
if (packet.ack == Ack_num) {
    Ack_num = (Ack_num + 1) % 256;
    cout << "Successful send and recv response  ack:" << int(packet.ack) << endl;
    confirmed++;
    last = (last + 1) % Window;
}
else {
    int dis = (int(packet.ack) - Ack_num) > 0 ? (int(packet.ack) - Ack_num) :
(int(packet.ack) + 256 - Ack_num);
    //忽略重复ACK
    if (packet.ack == Ack_num-1) {
        cout << "recv (same ACK)!!!!" << int(packet.ack) << endl;
        continue;
    }
    else if (dis < Window) {
        // 累计确认
        for (int temp = 0; temp <= dis; temp++) {
            cout << "===============累计确认===============" << endl;
            cout << "Successful send and recv response  ack:" << Ack_num + temp << endl;
            confirmed++;
            last = (last + 1) % Window;
        }
        Ack_num = (int(packet.ack) + 1) % 256;  //更新下次期待的ACK序号
    }
    else {  //数据校验出错、ack不合理  重发数据
        for (int temp = confirmed + 1; temp <= confirmed + Window && temp < packet_num;
temp++) {
            // 超时重新发送数据
            sendto(socketClient, Staging[temp % Window], sizeof(packet) +
Staging_len[cur % Window], 0, (sockaddr*)&servAddr, servAddrlen);
            cout << "出错重新发送暂存区内所有数据:  序列号为" << temp % 256 << endl;
        }
    }
}
```

9. 超时重传：

可靠传输设计需要满足信息通信双方对数据的交互的严苛把控。在实际实验设计上，采用超时重传机制以避免在网络不流畅情况下，保证信息交互的可靠性。

在GBN实验中，超时重传机制是面对丢包情况下关键的解决方法。如果某一个数据包超出规定的时间段未收到回传确认，那么便启动超时重传。重传窗口缓冲区中所有未被确认的分组。

```
time[cur % Window] = clock();    //在每条数据的时候记录发送时间  ——对应
...
while (recvfrom(socketClient, re_buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) <= 0) {
    int time_temp = last;
    if ((clock() - time[last]) / 1000 > 3)   //时间超过3秒 标记为超时重传
    {  //重传所有未被ack的数据
        //cout << last << "      " << time[last] << endl;
        for (int temp = confirmed + 1; temp <= confirmed + Window && temp < packet_num;
temp++) {
            // 超时重新发送数据
            sendto(socketClient, Staging[temp % Window], sizeof(packet) +
Staging_len[cur % Window], 0, (sockaddr*)&servAddr, servAddrlen);
            cout << "超时重新发送暂存区内所有数据:  序列号为" << temp % 256 << endl;
            time[time_temp++] = clock();
            time_temp /= Window;
        }
    }
}
```

三、核心函数&代码实现

1. 发送方与接收方协定单次传输数据大小&窗口大小

```
int Client_Server_Size(SOCKET& socketClient,SOCKADDR_IN& servAddr,int& servAddrlen, int
size, int Window,int*& A) {
    Packet_Header packet;
    char* buffer = new char[sizeof(packet)];  // 发送buffer
    // 接收客户端告知的缓冲区数据大小
    while (1) {
        if (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) == -1) {
            cout << "无法接收客户端发送的连接请求" << endl;
            return -1;
        }
        memcpy(&packet, buffer, sizeof(packet));
        if (packet.tag == SYN && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
            cout << "成功收到对方缓冲区大小信息" << endl;
            break;
        }
    }
    int client_size = packet.datasize;
    int client_win = packet.window;
    // 服务端告知客户端缓冲区大小信息
    packet.tag = ACK;
    packet.datasize = size;
    packet.window = Window;
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    if (sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen) == -1) {
        return -1;
    }
    cout << "成功告知对方缓冲区大小" << endl;

    A[0] = (size > client_size) ? client_size : size;
    A[1] = (Window > client_win) ? client_win : Window;
    return 1;
}
```

2. 发送方发送数据

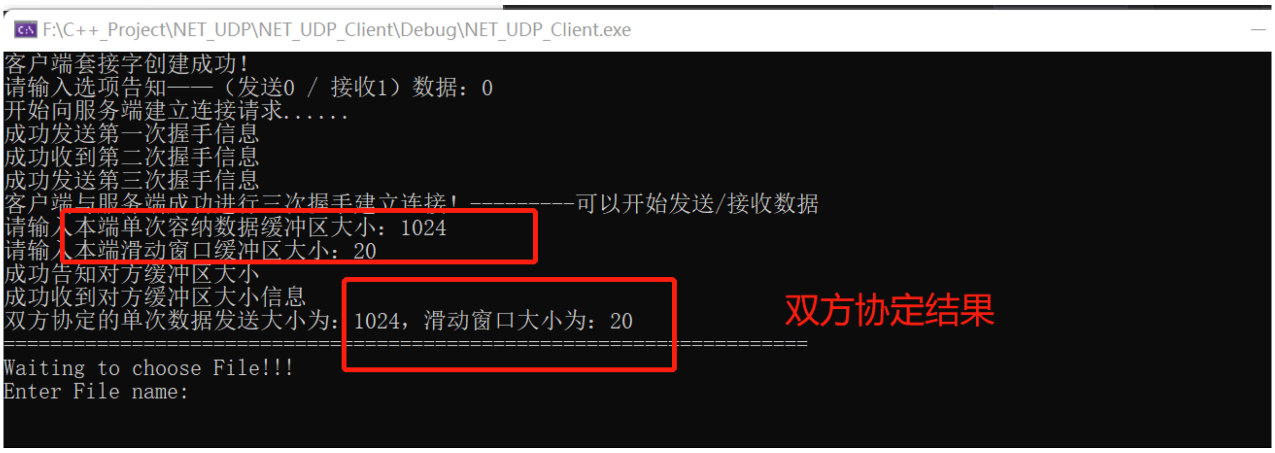
```
// 作为发送方发送数据
void Send_Message(SOCKET& socketClient, SOCKADDR_IN& servAddr, int& servAddrlen, char*
Message, int mes_size, int MAX_SIZE, int Window) {
    int packet_num = mes_size / (MAX_SIZE)+(mes_size % MAX_SIZE != 0);
    int Seq_num = 0;  //初始化序列号
    int Ack_num = 1;
    Packet_Header packet;
    int confirmed = -1;
    int cur = 0;
    int last = 0; //当前未收到确认的数据报序号
    char** Staging = new char* [Window];   //数据暂存区
    int* Staging_len = new int[Window];
    for (int i = 0; i < Window; i++) {
        Staging[i] = new char[sizeof(packet) + MAX_SIZE];
    }
    u_long mode = 1;
    ioctlsocket(socketClient, FIONBIO, &mode);  // 非阻塞模式
    while (confirmed < (packet_num - 1)) {  // 所有发送数据均被确认
        clock_t* time = new clock_t[Window];  //发送时间记录
        while (cur <= confirmed + Window && cur < packet_num) {
            int data_len = (cur == packet_num - 1 ? mes_size - (packet_num - 1) *
MAX_SIZE : MAX_SIZE);
            packet.tag = 0;
            packet.seq = Seq_num++;
            Seq_num = (Seq_num > 255 ? Seq_num - 256 : Seq_num);
            packet.datasize = data_len;
            Staging_len[cur % Window] = data_len;
            packet.window = Window - (cur - confirmed);  //剩余滑动窗口大小
            packet.checksum = 0;
            packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
            memcpy(Staging[cur % Window], &packet, sizeof(packet));
            char* mes = Message + cur * MAX_SIZE;
            memcpy(Staging[cur % Window] + sizeof(packet), mes, data_len);

            // 发送数据
            sendto(socketClient, Staging[cur % Window], sizeof(packet) + data_len, 0,
(sockaddr*)&servAddr, servAddrlen);
            // 记录每一次的发送时间
            time[cur % Window] = clock();
            cout << "Begin sending message...... datasize:" << data_len << " bytes!"
                << " seq:" << int(packet.seq) << " window:" << int(packet.window)
                << " checksum:" << int(packet.checksum) << "时间存储标号: " << last <<
endl;
            cur++;
        }

        char* re_buffer = new char[sizeof(packet)];
        while (recvfrom(socketClient, re_buffer, sizeof(packet), 0,
(sockaddr*)&servAddr, &servAddrlen) <= 0) {
            int time_temp = last;
            if ((clock() - time[last]) / 1000 > 2)   //时间超过2秒 标记为超时重传
            {  //重传所有未被ack的数据
                cout << "====================" << last << endl;
                for (int temp = confirmed + 1; temp <= confirmed + Window && temp <
packet_num; temp++) {
                    // 超时重新发送数据
                    sendto(socketClient, Staging[temp % Window], sizeof(packet) +
Staging_len[cur % Window], 0, (sockaddr*)&servAddr, servAddrlen);
                    cout << "超时重新发送暂存区内所有数据:  序列号为" << temp % 256 << "时
间存储标号: " << time_temp << endl;
                    time[time_temp] = clock();
                    time_temp = (time_temp + 1) % Window;
                }
            }
        }

        //成功收到响应
        memcpy(&packet, re_buffer, sizeof(packet));
        if ((compute_sum((WORD*)&packet, sizeof(packet)) != 0)) {  //数据校验出错
            continue;
        }
        //收到正确响应 滑动窗口
        if (packet.ack == Ack_num) {
            Ack_num = (Ack_num + 1) % 256;
            cout << "Successful send and recv response  ack:" << int(packet.ack) << "时
间存储标号: " << last << endl;
            confirmed++;
            time[last] = clock();
            last = (last + 1) % Window;
        }
        else {
            int dis = (int(packet.ack) - Ack_num) > 0 ? (int(packet.ack) - Ack_num) :
(int(packet.ack) + 256 - Ack_num);
            //忽略重复ACK
            if (packet.ack == (Ack_num + 256 - 1) % 256) {
                cout << "recv (same ACK)!!!!" << int(packet.ack) << endl;
                continue;
            }
            else if (dis < Window) {
                // 累计确认
                for (int temp = 0; temp <= dis; temp++) {
                    cout << "==============累计确认==============" << endl;
                    cout << "Successful send and recv response  ack:" << Ack_num + temp
<< endl;
                    confirmed++;
                    time[last] = clock();
                    last = (last + 1) % Window;
                }
                Ack_num = (int(packet.ack) + 1) % 256;  //更新下次期待的ACK序号
            }
            else {  //数据校验出错、ack不合理  重发数据
                for (int temp = confirmed + 1; temp <= confirmed + Window && temp <
packet_num; temp++) {
                    // 超时重新发送数据
                    sendto(socketClient, Staging[temp % Window], sizeof(packet) +
Staging_len[cur % Window], 0, (sockaddr*)&servAddr, servAddrlen);
                    cout << "出错重新发送暂存区内所有数据:  序列号为" << temp % 256 <<
endl;
                }
            }
        }
    }
    //发送结束标志
    packet.tag = OVER;
    char* buffer = new char[sizeof(packet)];
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr, servAddrlen);
    cout << "The end tag has been sent" << endl;
    clock_t start = clock();
    while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) <= 0) {
        if ((clock() - start) / 1000 > 1)   //时间超过1秒 标记为超时重传
        {
            sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen);
            cout << "数据发送超时，正在重传！！！" << endl;
            start = clock();
        }
    }
    mode = 0;
    ioctlsocket(socketClient, FIONBIO, &mode);  // 阻塞模式
    memcpy(&packet, buffer, sizeof(packet));
    if (packet.tag == OVER && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
        cout << "The end token was successfully received" << endl;
    }
    else {
        cout << "无法接收客户端回传" << endl;
    }
    return;
}
```

3. 接收方接收数据

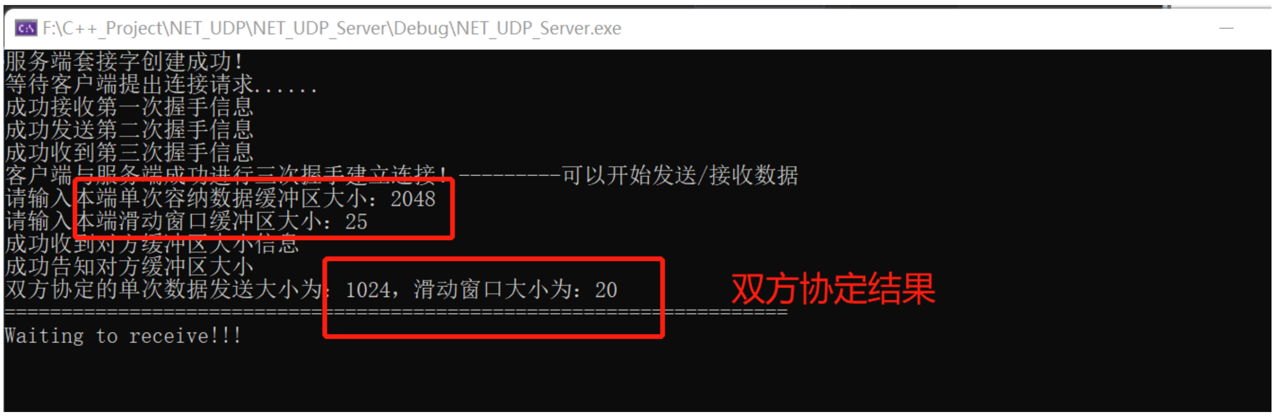
```
// 作为接收方接收数据
int Recv_Message(SOCKET& socketServer, SOCKADDR_IN& clieAddr, int& clieAddrlen, char*
Mes, int MAX_SIZE, int Window) {
    Packet_Header packet;
    char* buffer = new char[sizeof(packet) + MAX_SIZE];  // 接收buffer
    int ack = 1;  // 确认序列号
    int seq = 0;
    long F_Len = 0;     //数据总长
    int S_Len = 0;      //单次数据长度
    bool test_state = true;
    //接收数据
    while (1) {
        while (recvfrom(socketServer, buffer, sizeof(packet) + MAX_SIZE, 0,
(sockaddr*)&clieAddr, &clieAddrlen) <= 0);
        memcpy(&packet, buffer, sizeof(packet));

        if (((rand() % (255 - 1)) + 1) == 199 && test_state) {
            cout << int(packet.seq) << "随机超时重传测试" << endl;
            test_state = false;
            continue;
        }

        // END 全局结束
        if (packet.tag == END && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
            cout << "The all_end tag has been recved" << endl;
            return 999;
        }

        // 结束标记
        if (packet.tag == OVER && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
            cout << "The end tag has been recved" << endl;
            break;
        }

        // 接收数据
        if (packet.tag == 0 && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
            // 对接收数据加以确认，如果seq前的序列未被收到 则丢弃数据包 重传当前的最大ACK
            if (packet.seq != seq) {   // seq是当前按序接收的最高序号分组
                Packet_Header temp;
                temp.tag = 0;
                temp.ack = ack - 1;  //重传ACK
                temp.checksum = 0;
                temp.checksum = compute_sum((WORD*)&temp, sizeof(temp));
                memcpy(buffer, &temp, sizeof(temp));
                sendto(socketServer, buffer, sizeof(temp), 0, (sockaddr*)&clieAddr,
clieAddrlen);
                cout << "Send a resend flag (same ACK) to the client" << endl;
                continue;
            }

            // 成功收到数据
            S_Len = packet.datasize;
            cout << "Begin recving message...... datasize:" << S_Len << " bytes!" << "
flag:"
                << int(packet.tag) << " seq:" << int(packet.seq) << " checksum:" <<
int(packet.checksum) << endl;
            memcpy(Mes + F_Len, buffer + sizeof(packet), S_Len);
            F_Len += S_Len;

            // 返回 告知客户端已成功收到
            packet.tag = 0;
            packet.ack = ack++;
            packet.seq = seq++;
            packet.datasize = 0;
            packet.checksum = 0;
            packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
            memcpy(buffer, &packet, sizeof(packet));
            if (((rand() % (255 - 1)) + 1) != 187) {
                sendto(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
clieAddrlen);
            }
            else
                cout << "序列号: " << int(packet.seq) << " ========累计确认测试========"
<< endl;
            cout << "Successful recv and send response  ack : " << int(packet.ack) <<
endl;
            seq = (seq > 255 ? seq - 256 : seq);
            ack = (ack > 255 ? ack - 256 : ack);
        }
        else if (packet.tag == 0) {  // 校验和计算不正确、数据出错告知其重传
            Packet_Header temp;
            temp.tag = 0;
            temp.ack = ack - 1;  //按照接收乱序错误处理 重传ACK
            temp.checksum = 0;
            temp.checksum = compute_sum((WORD*)&temp, sizeof(temp));
            memcpy(buffer, &temp, sizeof(temp));
            sendto(socketServer, buffer, sizeof(temp), 0, (sockaddr*)&clieAddr,
clieAddrlen);
            cout << "Send a resend flag (errro ACK) to the client" << endl;
            continue;
        }
    }

    // 发送结束标志
    packet.tag = OVER;
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    sendto(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr, clieAddrlen);
    cout << "The end tag has been sent" << endl;
    return F_Len;
}
```

四、实际效果展示

- 本实验是建立在3-1的基础上，部分效果已在3-1报告中有所展示，不在本报告中重复描述。
- 打开服务端和客户端。在客户端应用界面输入本次客户端的功能定位： 发送方OR接收方 。双方在三次握手建立好连接后，首先需要协定好 单次数据发送大小 和 滑动窗口的大小 。



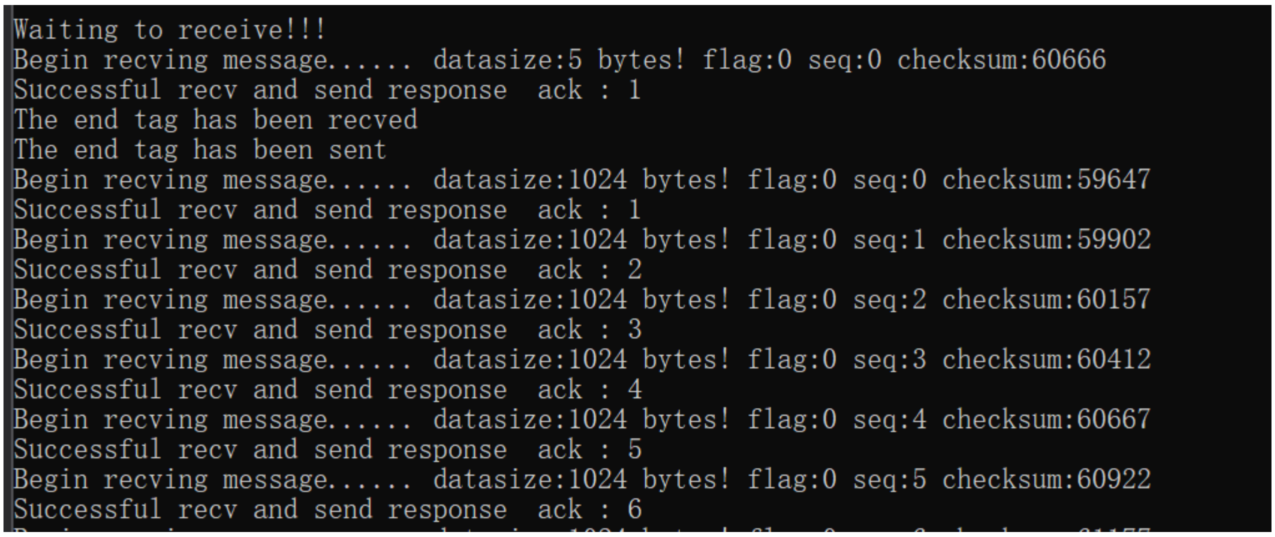
```
F:\C++_Project\NET_UDP\NET_UDP_Client\Debug\NET_UDP_Client.exe
客户端套接字创建成功!
请输入选项告知——（发送0 / 接收1）数据: 0
开始向服务端建立连接请求......
成功发送第一次握手信息
成功收到第二次握手信息
成功发送第三次握手信息
客户端与服务端成功进行三次握手建立连接！---------可以开始发送/接收数据
请输入本端单次容纳数据缓冲区大小: 1024
请输入本端滑动窗口缓冲区大小: 20
成功告知对方缓冲区大小
成功收到对方缓冲区大小信息
双方协定的单次数据发送大小为: 1024，滑动窗口大小为: 20
======================================================================
Waiting to choose File!!!
Enter File name:
```



```
F:\C++_Project\NET_UDP\NET_UDP_Server\Debug\NET_UDP_Server.exe
服务端套接字创建成功!
等待客户端提出连接请求......
成功接收第一次握手信息
成功发送第二次握手信息
成功收到第三次握手信息
客户端与服务端成功进行三次握手建立连接！---------可以开始发送/接收数据
请输入本端单次容纳数据缓冲区大小: 2048
请输入本端滑动窗口缓冲区大小: 25
成功收到对方缓冲区大小信息
成功告知对方缓冲区大小
双方协定的单次数据发送大小为: 1024，滑动窗口大小为: 20
======================================================================
Waiting to receive!!!
```

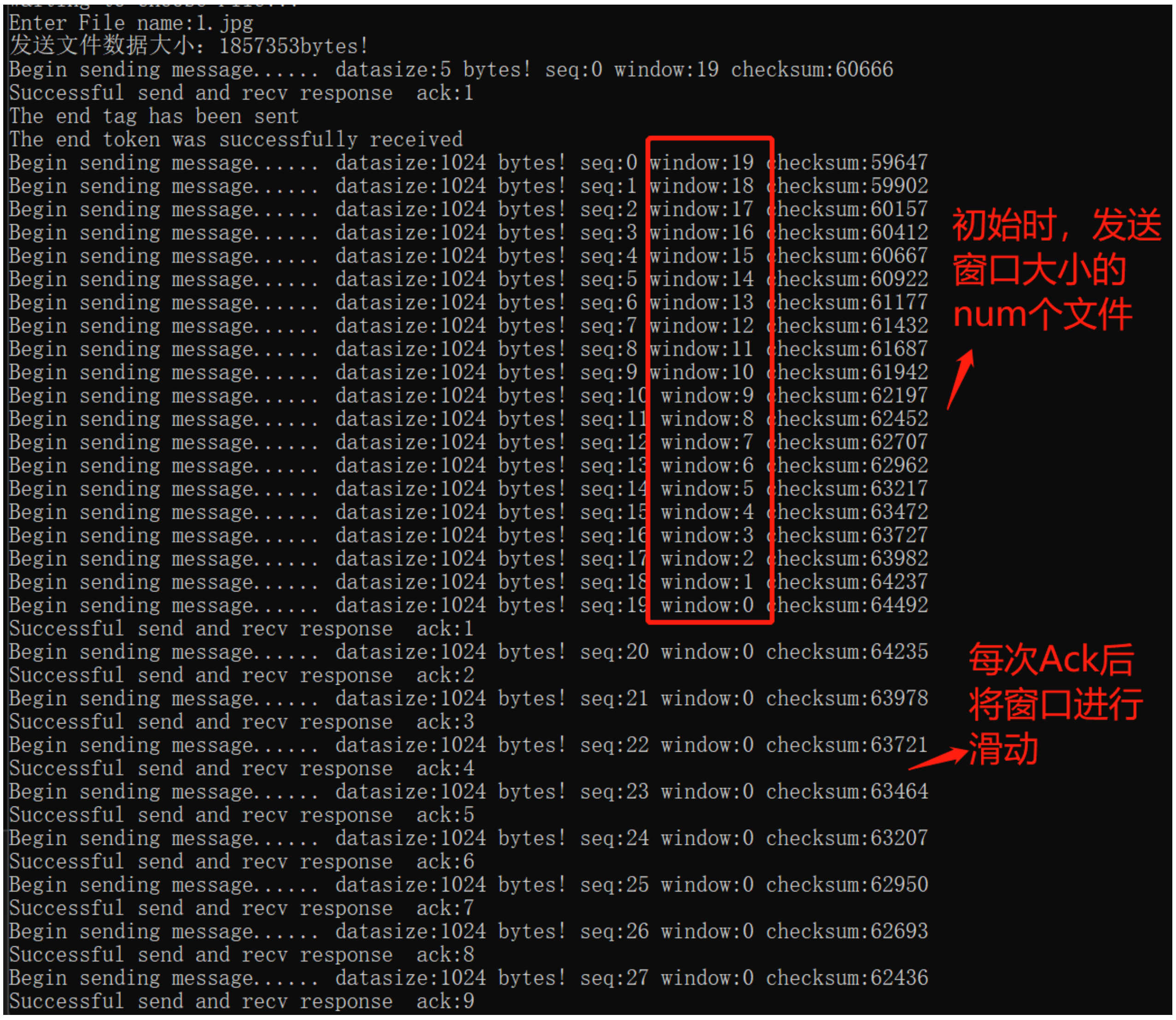
双方成功协定之后，由发送端选择需要传输的文件进行发送，观察发送方与接收方的日志输出结果。

- 接收方日志：逐条数据包进行Ack确认



```
Waiting to receive!!!
Begin recving message...... datasize:5 bytes! flag:0 seq:0 checksum:60666
Successful recv and send response  ack : 1
The end tag has been recved
The end tag has been sent
Begin recving message...... datasize:1024 bytes! flag:0 seq:0 checksum:59647
Successful recv and send response  ack : 1
Begin recving message...... datasize:1024 bytes! flag:0 seq:1 checksum:59902
Successful recv and send response  ack : 2
Begin recving message...... datasize:1024 bytes! flag:0 seq:2 checksum:60157
Successful recv and send response  ack : 3
Begin recving message...... datasize:1024 bytes! flag:0 seq:3 checksum:60412
Successful recv and send response  ack : 4
Begin recving message...... datasize:1024 bytes! flag:0 seq:4 checksum:60667
Successful recv and send response  ack : 5
Begin recving message...... datasize:1024 bytes! flag:0 seq:5 checksum:60922
Successful recv and send response  ack : 6
```

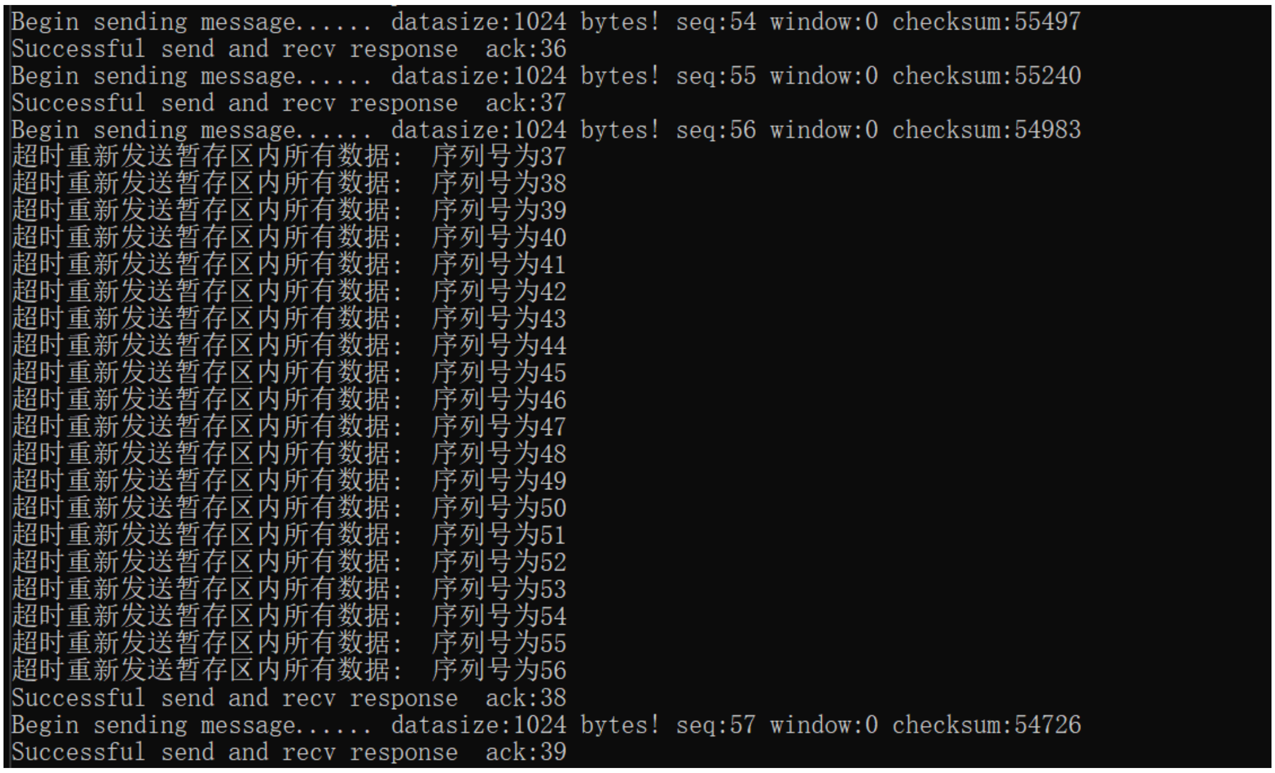
- 发送方日志：按照窗口的剩余大小进行数据包的发送



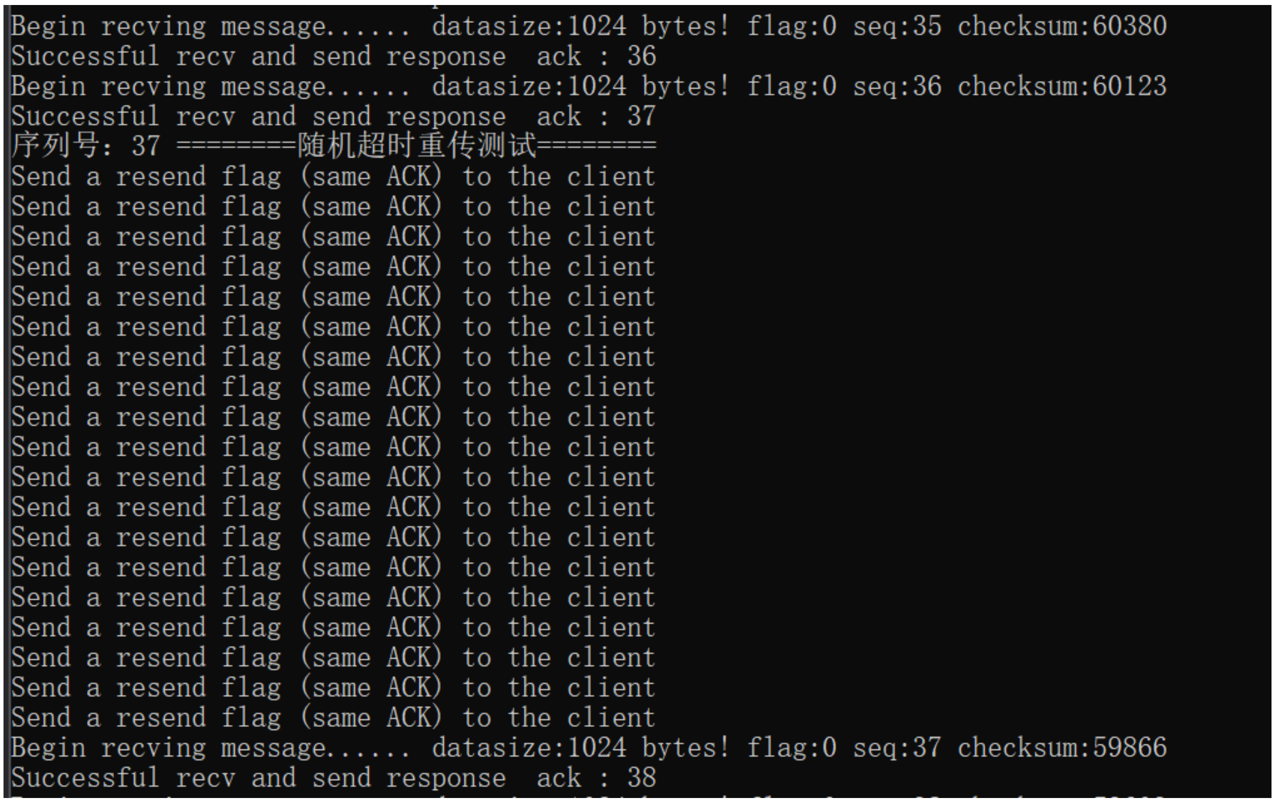
```
Enter File name:1.jpg
发送文件数据大小：1857353bytes!
Begin sending message...... datasize:5 bytes! seq:0 window:19 checksum:60666
Successful send and recv response  ack:1
The end tag has been sent
The end token was successfully received
Begin sending message...... datasize:1024 bytes! seq:0 window:19 checksum:59647
Begin sending message...... datasize:1024 bytes! seq:1 window:18 checksum:59902
Begin sending message...... datasize:1024 bytes! seq:2 window:17 checksum:60157
Begin sending message...... datasize:1024 bytes! seq:3 window:16 checksum:60412
Begin sending message...... datasize:1024 bytes! seq:4 window:15 checksum:60667
Begin sending message...... datasize:1024 bytes! seq:5 window:14 checksum:60922
Begin sending message...... datasize:1024 bytes! seq:6 window:13 checksum:61177
Begin sending message...... datasize:1024 bytes! seq:7 window:12 checksum:61432
Begin sending message...... datasize:1024 bytes! seq:8 window:11 checksum:61687
Begin sending message...... datasize:1024 bytes! seq:9 window:10 checksum:61942
Begin sending message...... datasize:1024 bytes! seq:10 window:9 checksum:62197
Begin sending message...... datasize:1024 bytes! seq:11 window:8 checksum:62452
Begin sending message...... datasize:1024 bytes! seq:12 window:7 checksum:62707
Begin sending message...... datasize:1024 bytes! seq:13 window:6 checksum:62962
Begin sending message...... datasize:1024 bytes! seq:14 window:5 checksum:63217
Begin sending message...... datasize:1024 bytes! seq:15 window:4 checksum:63472
Begin sending message...... datasize:1024 bytes! seq:16 window:3 checksum:63727
Begin sending message...... datasize:1024 bytes! seq:17 window:2 checksum:63982
Begin sending message...... datasize:1024 bytes! seq:18 window:1 checksum:64237
Begin sending message...... datasize:1024 bytes! seq:19 window:0 checksum:64492
Successful send and recv response  ack:1
Begin sending message...... datasize:1024 bytes! seq:20 window:0 checksum:64235
Successful send and recv response  ack:2
Begin sending message...... datasize:1024 bytes! seq:21 window:0 checksum:63978
Successful send and recv response  ack:3
Begin sending message...... datasize:1024 bytes! seq:22 window:0 checksum:63721
Successful send and recv response  ack:4
Begin sending message...... datasize:1024 bytes! seq:23 window:0 checksum:63464
Successful send and recv response  ack:5
Begin sending message...... datasize:1024 bytes! seq:24 window:0 checksum:63207
Successful send and recv response  ack:6
Begin sending message...... datasize:1024 bytes! seq:25 window:0 checksum:62950
Successful send and recv response  ack:7
Begin sending message...... datasize:1024 bytes! seq:26 window:0 checksum:62693
Successful send and recv response  ack:8
Begin sending message...... datasize:1024 bytes! seq:27 window:0 checksum:62436
Successful send and recv response  ack:9
```

- 代码中添加了关于接收方无法接受到数据的模拟测试，观察超时情况下数据的重传

```
if (((rand() % (255 - 1)) + 1) == 199) {
    cout << "序列号: " << int(packet.seq) << " ========随机超时重传测试========" << endl;
    continue;
}
```



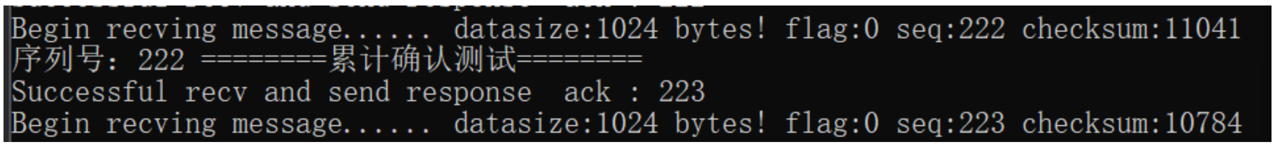
```
Begin sending message...... datasize:1024 bytes! seq:54 window:0 checksum:55497
Successful send and recv response  ack:36
Begin sending message...... datasize:1024 bytes! seq:55 window:0 checksum:55240
Successful send and recv response  ack:37
Begin sending message...... datasize:1024 bytes! seq:56 window:0 checksum:54983
超时重新发送暂存区内所有数据:  序列号为37
超时重新发送暂存区内所有数据:  序列号为38
超时重新发送暂存区内所有数据:  序列号为39
超时重新发送暂存区内所有数据:  序列号为40
超时重新发送暂存区内所有数据:  序列号为41
超时重新发送暂存区内所有数据:  序列号为42
超时重新发送暂存区内所有数据:  序列号为43
超时重新发送暂存区内所有数据:  序列号为44
超时重新发送暂存区内所有数据:  序列号为45
超时重新发送暂存区内所有数据:  序列号为46
超时重新发送暂存区内所有数据:  序列号为47
超时重新发送暂存区内所有数据:  序列号为48
超时重新发送暂存区内所有数据:  序列号为49
超时重新发送暂存区内所有数据:  序列号为50
超时重新发送暂存区内所有数据:  序列号为51
超时重新发送暂存区内所有数据:  序列号为52
超时重新发送暂存区内所有数据:  序列号为53
超时重新发送暂存区内所有数据:  序列号为54
超时重新发送暂存区内所有数据:  序列号为55
超时重新发送暂存区内所有数据:  序列号为56
Successful send and recv response  ack:38
Begin sending message...... datasize:1024 bytes! seq:57 window:0 checksum:54726
Successful send and recv response  ack:39
```



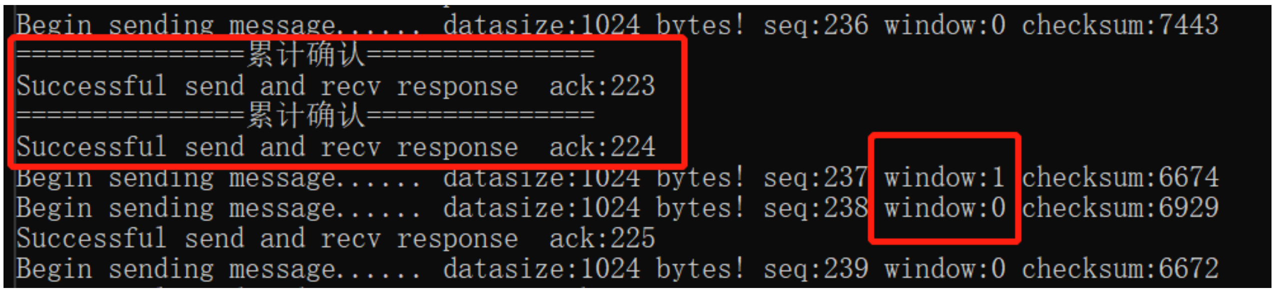
```
Begin recving message...... datasize:1024 bytes! flag:0 seq:35 checksum:60380
Successful recv and send response  ack : 36
Begin recving message...... datasize:1024 bytes! flag:0 seq:36 checksum:60123
Successful recv and send response  ack : 37
序列号: 37 ========随机超时重传测试========
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Send a resend flag (same ACK) to the client
Begin recving message...... datasize:1024 bytes! flag:0 seq:37 checksum:59866
Successful recv and send response  ack : 38
```

- 代码中添加了关于接收方回传的ACK确认数据包丢失的情况，观察发送方的累计确认机制对现象的处理

```
memcpy(buffer, &packet, sizeof(packet));
if (((rand() % (255 - 1)) + 1) != 187) {
    sendto(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr, clieAddrlen);
}
else  // 模拟接收方回传的ACK确认数据包丢失
    cout << "序列号: " << int(packet.seq) << " ========累计确认测试========" << endl;
// 接收方并不知道发送方会接收不到ACK确认数据报
cout << "Successful recv and send response  ack : " << int(packet.ack) << endl;
seq = (seq > 255 ? seq - 256 : seq);
ack = (ack > 255 ? ack - 256 : ack);
```

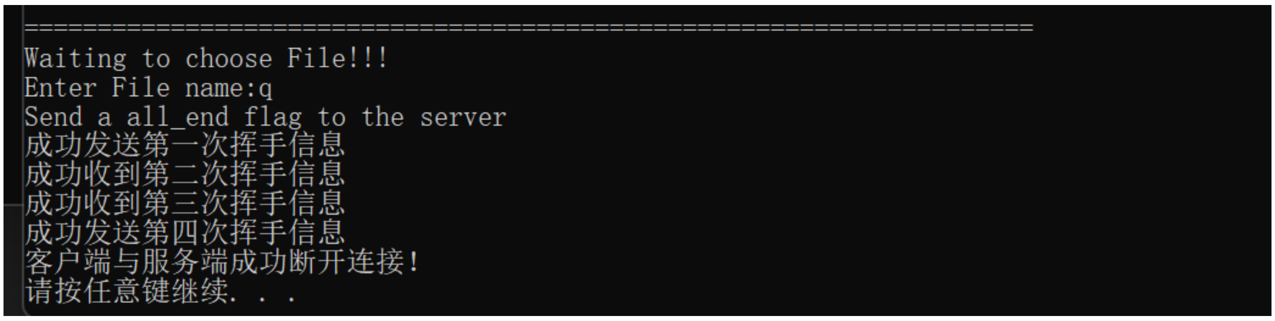


```
Begin recving message...... datasize:1024 bytes! flag:0 seq:222 checksum:11041
序列号: 222 ========累计确认测试========
Successful recv and send response  ack : 223
Begin recving message...... datasize:1024 bytes! flag:0 seq:223 checksum:10784
```

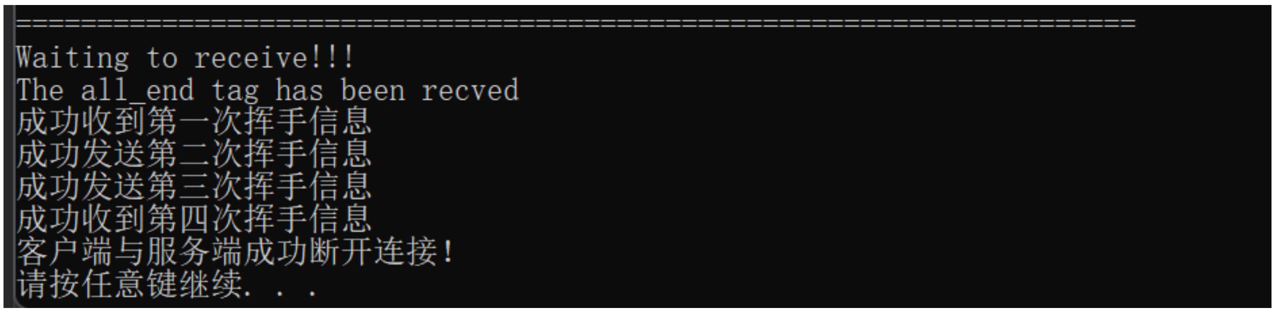


```
Begin sending message...... datasize:1024 bytes! seq:236 window:0 checksum:7443
===============累计确认===============
Successful send and recv response  ack:223
===============累计确认===============
Successful send and recv response  ack:224
Begin sending message...... datasize:1024 bytes! seq:237 window:1 checksum:6674
Begin sending message...... datasize:1024 bytes! seq:238 window:0 checksum:6929
Successful send and recv response  ack:225
Begin sending message...... datasize:1024 bytes! seq:239 window:0 checksum:6672
```

- 程序可以一直运行，发送文件数据。在用户主动按下q键的情况下，数据文件传输结束，进行断联过程，四次挥手双方断开连接

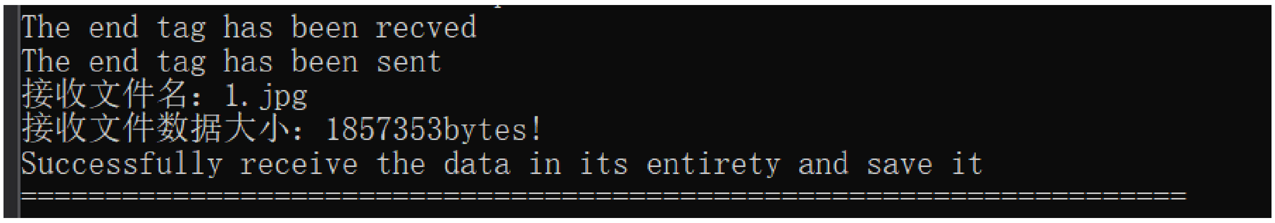


```
====================================================================
Waiting to choose File!!!
Enter File name:q
Send a all_end flag to the server
成功发送第一次挥手信息
成功收到第二次挥手信息
成功收到第三次挥手信息
成功发送第四次挥手信息
客户端与服务端成功断开连接!
请按任意键继续. . .
```



```
=====================================================================
Waiting to receive!!!
The all_end tag has been recved
成功收到第一次挥手信息
成功发送第二次挥手信息
成功发送第三次挥手信息
成功收到第四次挥手信息
客户端与服务端成功断开连接!
请按任意键继续. . .
```

- 发送方使用fstrean二进制方式打开文件，进行传输。接收方在收到所有内容之后，使用fstrean二进制方式将其保存写入。



```
The end tag has been recved
The end tag has been sent
接收文件名: 1.jpg
接收文件数据大小: 1857353bytes!
Successfully receive the data in its entirety and save it
=====================================================================
```

五、实验总结

基于UDP服务实现可靠传输实验第一部分的基础上，成功将停等机制替换成基于滑动窗口的流量控制机制，支持累计确认，并完成给定测试文件的传输。

除却实现规定实验功能，作者还额外实现如下功能：

1. 双向传输：客户端与服务端双方都可以作为发送方，另一端作为接收方。这样的功能定义通过客户端在三次握手的第三次握手数据包发送时将`label`标签在双方之间进行传递，以此双方进行确定本次应用双方的角色。
2. 单次发送数据大小&滑动窗口大小协商确定：客户端与服务端在文件数据`send&recv`之前，先进行一次协商。客户端与服务端各自在数据帧中告知对方相关数据，双方按照取`min`的方法进行协定。
3. 程序不会在完成一次文件数据传输后就退出程序，而是在用户主动按`q`键后，才会`end`程序。
4. 在实验的过程中，主动添加超时重传的测试环节（模拟发送方丢包，接收方无法按序接收造成超时未收到确认的情况），测试发送方与接收方遇到超时情况下是否能够正确处理。
5. 在实验的过程中，主动添加累计确认的测试环节（模拟接收方回传`ACK`的数据包中发生丢包，发送方收到的`ACK`表明接收方已正确接收到序号为 `n` 的以前且包括 `n` 在内的所有分组。）测试发送方与接收方在接收方回传`Ack`发生丢包时是否能够正确处理。
6. 本实验中手动添加了关于发送端与接收端双方均出现丢包的情形，丢包后双端的处理结果在上述实验效果图中均有所展示。路由器设置丢包率的实验结果在lab6中会进行全面的性能分析，就不在此做展示。

在后续进一步的实验设计中，希望能够实现更加安全、可靠、快速的文件数据传输。