任务3-2：

> 在任务3-1的基础上，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。

协议设计

使用队列模拟滑动窗,滑动窗的nextseqnum设为数据包的序号,数据包的设置与实验一一致:

首先使用发送内容和最大数据包长度求得所有需要发送的包的数量,然后发送时,在裁剪好的数据包的首部加上3或4位分别表示校验和,是否是最后一位,序号和包长度(在当前包为最后一个数据包时生效).

滑动窗的设计使用队列,客户端收到的ACK在ackwindow中进行标识.

```
//滑动窗中保存时间和序号
queue<pair<int,int>> sw;
static int sendbase = 1;
int nextseqnum = sendbase;
bool ackwindow[256];
```

数据发送

见注释,如果发送窗存在空闲窗口,则发送数据包并使用滑动窗压入发送出去的包的发送时间和序号,且在ackwindow中标注此包为已发送.

接收到ack则前移sendbase,超时则重置滑动窗和包序号,实现重传数据包

```
//以打包发送一个非末尾数据包举例, 数据包的序号设置为nextseqnum
send_buffer = new char[content_len + 3];
        send_buffer[1] = NOTLAST;
        //序号设置为nextseqnum
        send_buffer[2] = nextseqnum;
        for (int i = 3; i < content_len + 3; i++)
        {
            send_buffer[i] = message[i - 3];
        }
        send_buffer[0] = checksum(send_buffer + 1, content_len + 2);
        send_buffer_len = content_len + 3;

//滑动窗的使用
while (1) {
        if (curpkt_num== num)
            //如果当前待发包序号为裁剪得到的所有包总数,则说明发送完成
            break;
        if(sw.size() < Window_Len && sent_pktnum < num)
        {//窗口有空闲且发出包数量小于包总数
            send(message + sent_pktnum * MAXLEN, sent_pktnum == num - 1 ?
content_len % MAXLEN : MAXLEN, nextseqnum % 256, sent_pktnum == num - 1);
            //压入当前发送的包的时间和序号
            sw.push(make_pair(clock(), nextseqnum % 256));
            ackwindow[nextseqnum%256] = true;
            nextseqnum++;
            sent_pktnum++;
        }
        char rev_ack[3];//接收ACK
        int addr_len = sizeof(clientAddr);
        int recvsize = recvfrom(client, rev_ack, 3, 0, (sockaddr*)&serverAddr,
&addr_len);
        if (recvsize&& checksum(rev_ack,3)==0 &&rev_ack[1]==ACK &&
ackwindow[(unsigned char)rev_ack[2]])//收到ACK且前一个数据包的ACK已经收到
        {
            while (sw.front().second != (unsigned char)rev_ack[2])
            {
                sendbase++;
                curpkt_num++;
                ackwindow[sw.front().second] = 0;
                sw.pop();
            }
            sendbase++;
            curpkt_num++;
            ackwindow[sw.front().second] = 0;
            sw.pop();
        }
        else {
            if (clock() - sw.front().first > TIMEOUT) {
                //超时 重置滑动窗 重传
                nextseqnum = sendbase;
                sent_pktnum -= sw.size();
                while (sw.size() != 0)
                    sw.pop();
            }
        }
    }
```

累积确认

服务端收到数据包时,会检查其序号是否是上一个收到的包的序号,是的话,跳过此次收取,否则刷新序号.

```
void recv(char *message,int &len_recv){
    char recv[MAXLEN + 4];
    int len_tmp = sizeof(clientAddr);
    static char last_seqnum = 0;
    len_recv = 0;
    while (true)
    {
        while (true)
        {
            memset(recv,0,sizeof(recv));
            while (recvfrom(server, recv, MAXLEN + 4, 0, (sockaddr *)
&clientAddr, &len_tmp) == SOCKET_ERROR);
            char send[3];
            if (checksum(recv, MAXLEN + 4) == 0)
            {
                send[1] = ACK;
                send[2] = recv[2];
                send[0] = checksum(send + 1, 2);
                sendto(server, send, 3, 0, (sockaddr *) &clientAddr,
sizeof(clientAddr));
                break;
            }
            else
            {
                send[1] = NAK;
                send[2] = recv[2];
                send[0] = checksum(send + 1, 2);
                sendto(server, send, 3, 0, (sockaddr *) &clientAddr,
sizeof(clientAddr));
                cout << "NAK" << endl;
                continue;
            }
        }
        if (last_seqnum == recv[2])
            continue;
        last_seqnum = recv[2];
        if (LAST == recv[1])
        {
            for (int i = 4; i < strlen(recv); i++)
                message[len_recv++] = recv[i];
            break;
        }
        else
        {
            for (int i = 3; i < MAXLEN + 3; i++)
                message[len_recv++] = recv[i];
        }
    }
}
```