Lab3-2——基于UDP服务设计可靠传输协议并编程实现

学号：2212452
姓名：孟启轩
专业：计算机科学与技术

一、实验要求

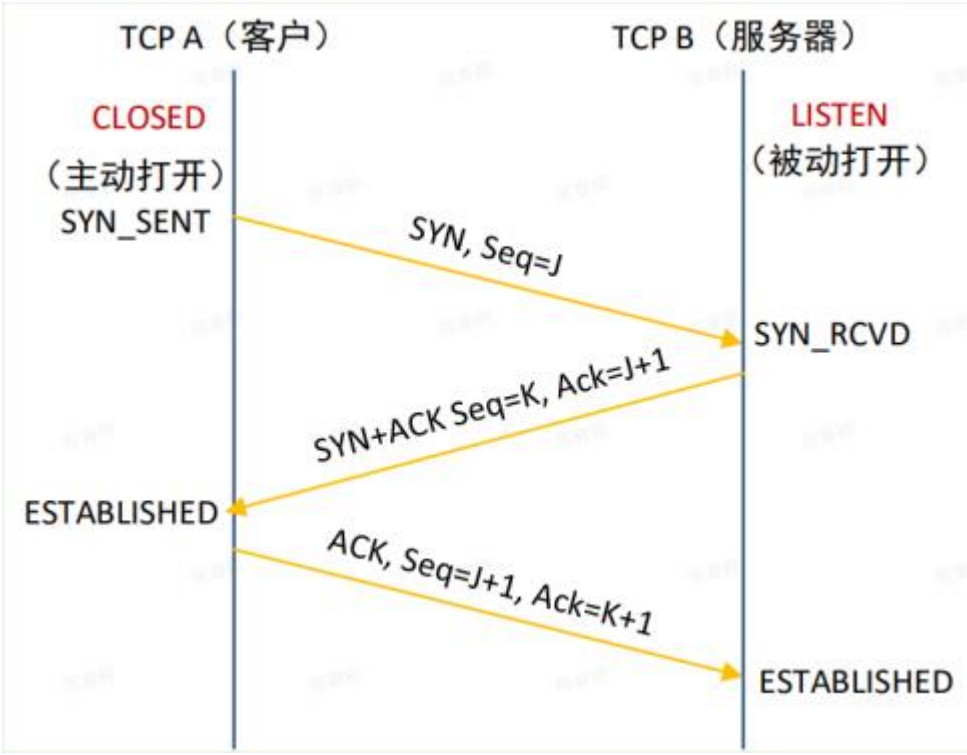
在实验3-1的基础上，将停等机制改成基于滑动窗口的流量控制机制，采用固定窗口大小，支持累积确认，完成给定测试文件的传输。

二、实验设计

(一)原理探究

> 三次握手

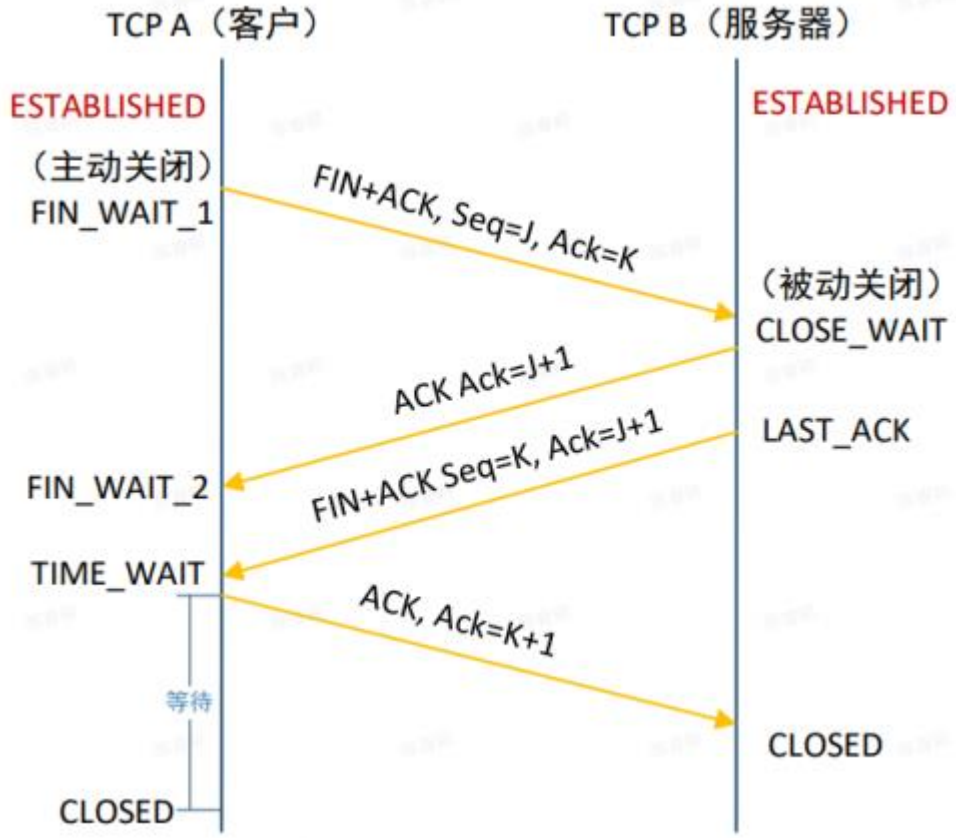
1. 第一次握手：客户端发送SYN包(seq=x)到服务器，并进入SYN_SEND状态，等待服务器确认
2. 第二次握手：服务器收到SYN包，必须确认客户的SYN(ack=x+1)，同时自己也发送一个SYN包(seq=y)，即SYN+ACK包，此时服务器进入SYN_RECV状态
3. 第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=y+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP连接都将被一直保持下去。



> 四次挥手

1. 第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN_WAIT_1状态。
2. 第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE_WAIT状态。

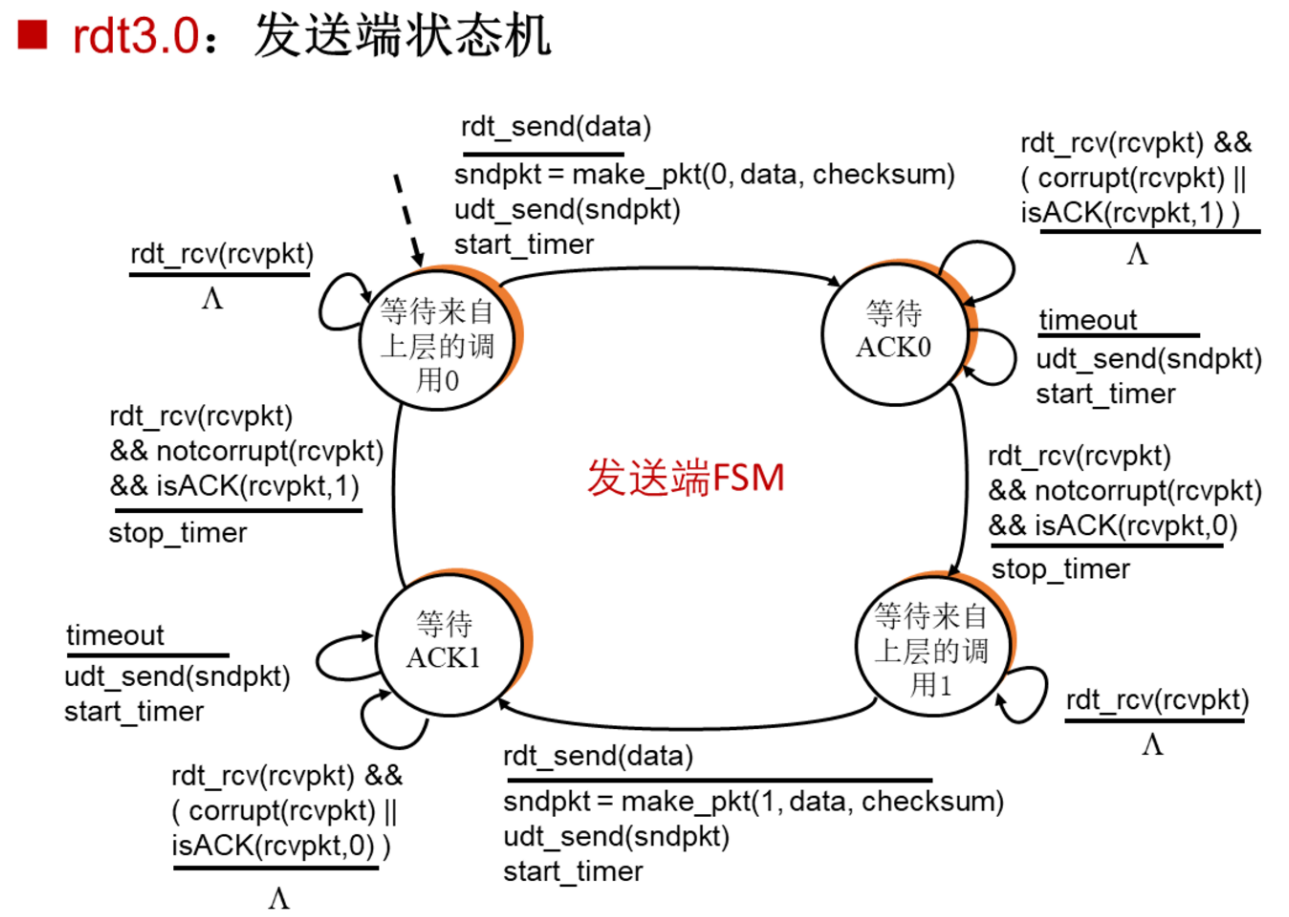
3. 第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST ACK状态。
4. 第四次挥手：Client收到FIN后，Client进入TIME_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。



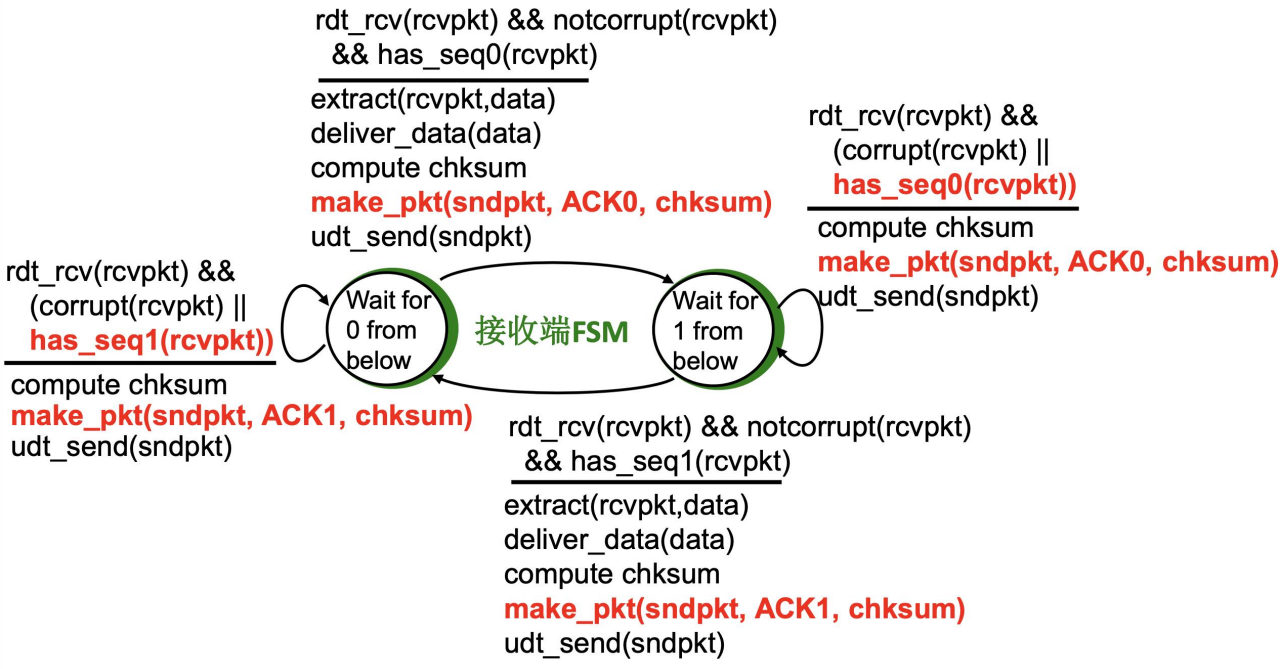
数据传输

发送端和接收端的接收机都采用rdt3.0的设计原则。

- 发送端的有限状态机：



- 接收端的有限状态机：



序列号与确认应答号

数据报的传输顺序通过序列号来保障，以确保数据传输的可靠性。每个响应包中也包含一个序列号，用于表明接收方已准备好处理对应的包。在传输数据报时，这个数据报会被放入重发队列，同时启动一个计时器。如果收到该数据包的确认信息，发送方会将此数据包从队列中删除。如果在计时器超时后仍未收到确认信息，则需

要重新发送该数据包。此外，通过数据分段中的序列号，可以确保所有传输的数据按正确的顺序进行重组，从而保证数据传输的完整性。

> 差错检测

差错检测（error detection），是指在发送的码序列（码字）中加入适当的冗余度以使得接收端能够发现传输中是否发生差错的技术。除了用于通信外，差错检测技术也广泛用于信息存储中。

> 超时重传

超时重传指的是在发送数据报文段后开始计时，设置一个等待确认应答到来的那个时间间隔。如果超过这个时间间隔，仍未收到对方发来的确认应答，发送端将进行数据重传。

> UDP协议

UDP是User Datagram Protocol的简称,中文名是用户数据报协议,是OSI参考模型中的传输层协议,它是一种无连接的传输层协议,提供面向事务的简单不可靠信息传送服务。

(二)协议设计

本程序中的相关设计如下：

> 三次握手建立连接

1. 客户端首先向服务端发送第一次握手请求(0x1)。
2. 服务端收到第一次握手请求后，向客户端发送第二次握手请求(0x2)。
3. 客户端收到第二次握手请求后，向服务端发送第三次握手请求(0x3)。
4. 服务端收到第三次握手请求后，握手成功，从而建立起通信连接。

> 四次挥手断开连接

1. 客户端首先向服务端发送第一次挥手请求(0x4)。
2. 服务端收到第一次挥手请求后，向客户端发送第二次挥手请求(0x5)。
3. 客户端收到第二次挥手请求后，挥手成功，通信正式断开

> 数据传输

- 客户端将文件数据划分为多个数据包，并通过UDP协议将这些数据包发送给服务器。每个数据包包含数据头部（包括数据大小、校验和、标志等信息）以及实际的文件数据。服务器在收到数据包后，会进行校验和确认，并根据数据包的标志进行相应的处理。
- 如果接收到的是正常的数据包，服务器会向客户端发送确认数据包。如果接收到的是最后一个数据包，服务器将发送一个带有结束标志（0x7）的确认数据包给客户端。

> 超时重传

- 在初始化函数中，设置了超时时间（MAX_WAIT_TIME）。
- 在Send_Message函数和初始化函数中，使用计时器来检测是否超时，并在超时后进行相应的重传操作。

> 差错检验

- 在Header类中，使用sum字段来存储校验和。
- 在check_sum函数中，实现了报文的校验和计算和设置。

- 而在Send_Message函数中，在构建发送的报文时计算校验和，并在接收端对收到的报文进行校验和的验证。

> 滑动窗口与GO BACK N协议

在后面（四）进行详细解释。

> 累积确认

累积确认是一种简单高效的ACK机制，主要特点是每个ACK都隐式确认了之前所有连续接收到的数据包。累积确认可以减少ACK数量，简化接收端设计，与GBN结合使用，帮助发送端管理重传。

三、设计实现

(一)校验和计算

- 发送方生成检验和

  1. 对数据报进行校验和运算的数据分成若干个16位的位串，每个位串看成一个二进制数
  2. 将首部中的检验和字段置为全0，该字段也参与检验和运算
  3. 对这些16位的二进制数进行1的补码和运算，累加的结果再取反码即生成了检验码。将检验码放入检验和字段中
  4. 其中1的补码和运算，即带循环进位的加法，最高位有进位应循环进到最低位

- 接收方校验检验和

  1. 接收方将接收的数据报按发送方的同样的方法进行1的补码和运算，累加的结果取反码
  2. 如果第一步的结果为0，表示传输正确
  3. 否则，说明传输出现差错

```
// 服务端计算校验和的函数，用于差错检测
unsigned short check_sum(unsigned short *message, int size)
{
    //int count = (size + 1) / 2;//客户端
    int count = (size + 1) / sizeof(unsigned short);
    unsigned short *buf = (unsigned short *)malloc(size + 1);
    memset(buf, 0, size + 1);
    memcpy(buf, message, size);
    unsigned long sum = 0;
    while (count--)
    {
        sum += *buf++;
        if (sum & 0xffff0000)
        {
            sum &= 0xffff;
            sum++;
        }
    }
    return ~(sum & 0xffff);
}
```

(二)数据报文格式

Header结构体负责管理数据包的必要元信息，比如大小、校验和、标志和序列号，以确保数据传输的可靠性和顺序。

- 数据包的头部信息

```
struct HEADER
{
    unsigned short sum = 0;
    unsigned short datasize = 0;
    unsigned char flag = 0;
    unsigned char SEQ = 0;
};
```

- 报文头部：长度为48位
- 1-16位为数据长度，记录数据的大小
- 17-32位为校验和，用于检验传输的正确性
- 33-40位为标志位，用于识别FIN，ACK，SYN等标志
- 41-48位为序列号，范围在0-255

(三)握手与挥手

- 客户端和服务端流程比较类似，我这里只给出客户端代码。

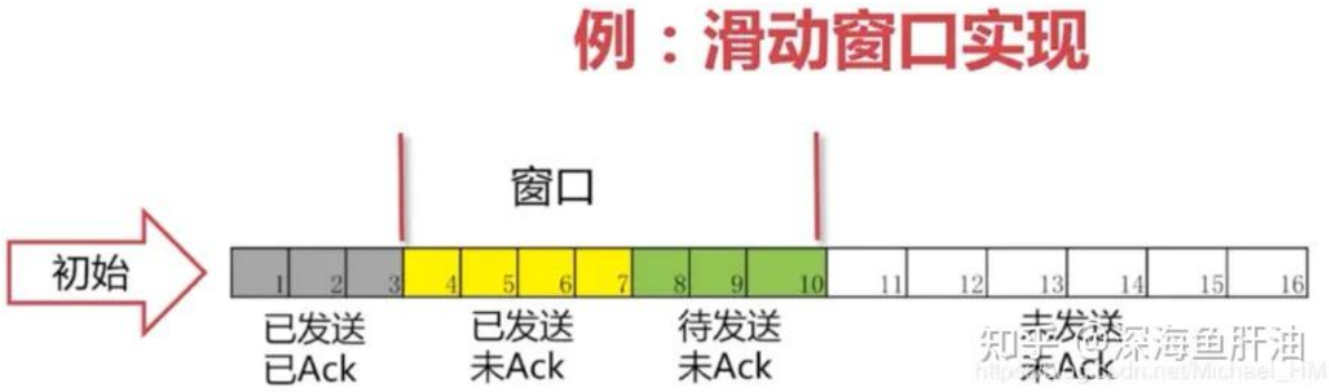
```
//握手
int Shake_hand(SOCKET &socketClient, SOCKADDR_IN &server_addr, int
&server_addr_len)
{
    HEADER header;
    char *buffer = new char[sizeof(header)];
    unsigned short sum;
    auto sendHeader = [&]() {
        header.sum = check_sum((unsigned short *)&header, sizeof(header));
        memcpy(buffer, &header, sizeof(header));
        return sendto(socketClient, buffer, sizeof(header), 0, (sockaddr
*)&server_addr, server_addr_len);
    };

    // 第一次握手请求
    header.flag = SYN;
    if (sendHeader() == SOCKET_ERROR) return SOCKET_ERROR;
    cout << "发送第一次握手请求" << endl;
    clock_t start = clock();
    unsigned long mode = 1;
    ioctlsocket(socketClient, FIONBIO, &mode);
    // 接收第二次握手响应
    while (recvfrom(socketClient, buffer, sizeof(header), 0, (sockaddr
*)&server_addr, &server_addr_len) <= 0)
    {
```

```
        if (clock() - start > MAX_WAIT_TIME) // 超时，重新传输第一次握手
        {
            header.flag = SYN;
            if (sendHeader() == SOCKET_ERROR) return SOCKET_ERROR;
            cout << "第一次握手超时，开始重传..." << endl;
            start = clock();
        }
    }
    memcpy(&header, buffer, sizeof(header));
    if (header.flag != ACK || check_sum((unsigned short *)&header, sizeof(header))
!= 0)
    {
        return SOCKET_ERROR;
    }
    cout << "收到第二次握手请求" << endl;
    // 第三次握手确认
    header.flag = ACK_SYN;
    if (sendHeader() == SOCKET_ERROR) return SOCKET_ERROR;
    cout << "发送第三次握手请求" << endl;
    cout << "服务器已连接！" << endl;
    delete[] buffer; // 清理分配的内存
    return 1;
}

//挥手
int Wave_hand(SOCKET &socketClient, SOCKADDR_IN &server_addr, int
&server_addr_len)
{
    HEADER header;
    char *buffer = new char[sizeof(header)];
    auto sendHeader = [&]() {
        header.sum = check_sum((unsigned short *)&header, sizeof(header));
        memcpy(buffer, &header, sizeof(header));
        return sendto(socketClient, buffer, sizeof(header), 0, (sockaddr
*)&server_addr, server_addr_len);
    };

    // 发送挥手请求
    header.flag = FIN;
    if (sendHeader() == SOCKET_ERROR) {
        delete[] buffer; // 清理分配的内存
        return SOCKET_ERROR;
    }
    cout << "发送第一次挥手请求" << endl;

    clock_t start = clock();
    unsigned long mode = 1;
    ioctlsocket(socketClient, FIONBIO, &mode);

    // 接收挥手响应
    while (recvfrom(socketClient, buffer, sizeof(header), 0, (sockaddr
*)&server_addr, &server_addr_len) <= 0)
    {
        if (clock() - start > MAX_WAIT_TIME) // 超时，重新传输挥手请求
```

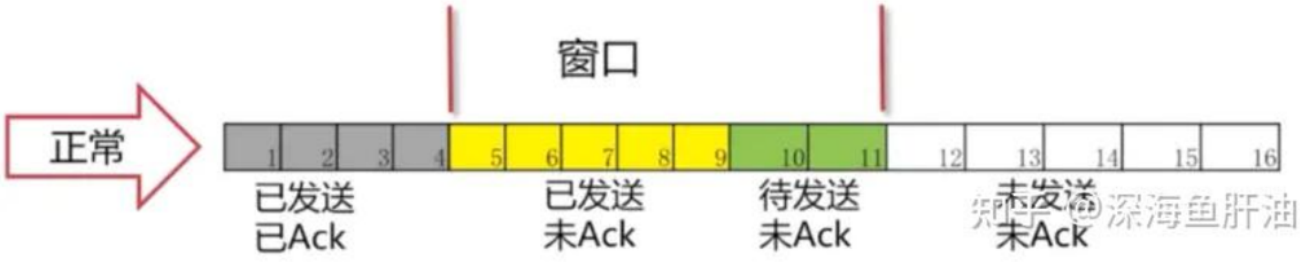
```
        {
            header.flag = FIN;
            if (sendHeader() == SOCKET_ERROR) {
                delete[] buffer; // 清理分配的内存
                return SOCKET_ERROR;
            }
            start = clock();
            cout << "第一次挥手超时，开始重传..." << endl;
        }
    }
    memcpy(&header, buffer, sizeof(header));
    if (header.flag != ACK || check_sum((unsigned short *)&header, sizeof(header))
!= 0) {
        delete[] buffer; // 清理分配的内存
        return SOCKET_ERROR;
    }
    cout << "收到第二次挥手请求" << endl;
    cout << "断开连接！" << endl;
    delete[] buffer; // 清理分配的内存
    return 1;
}
```

(四)滑动窗口（Go Back N 协议）

滑动窗口是一种用于流量控制的技术。在传统的网络通信中，客户端通常不考虑网络的拥塞情况，直接发送数据，这可能导致中间节点阻塞和数据包丢失。滑动窗口机制被引入以解决这一问题。

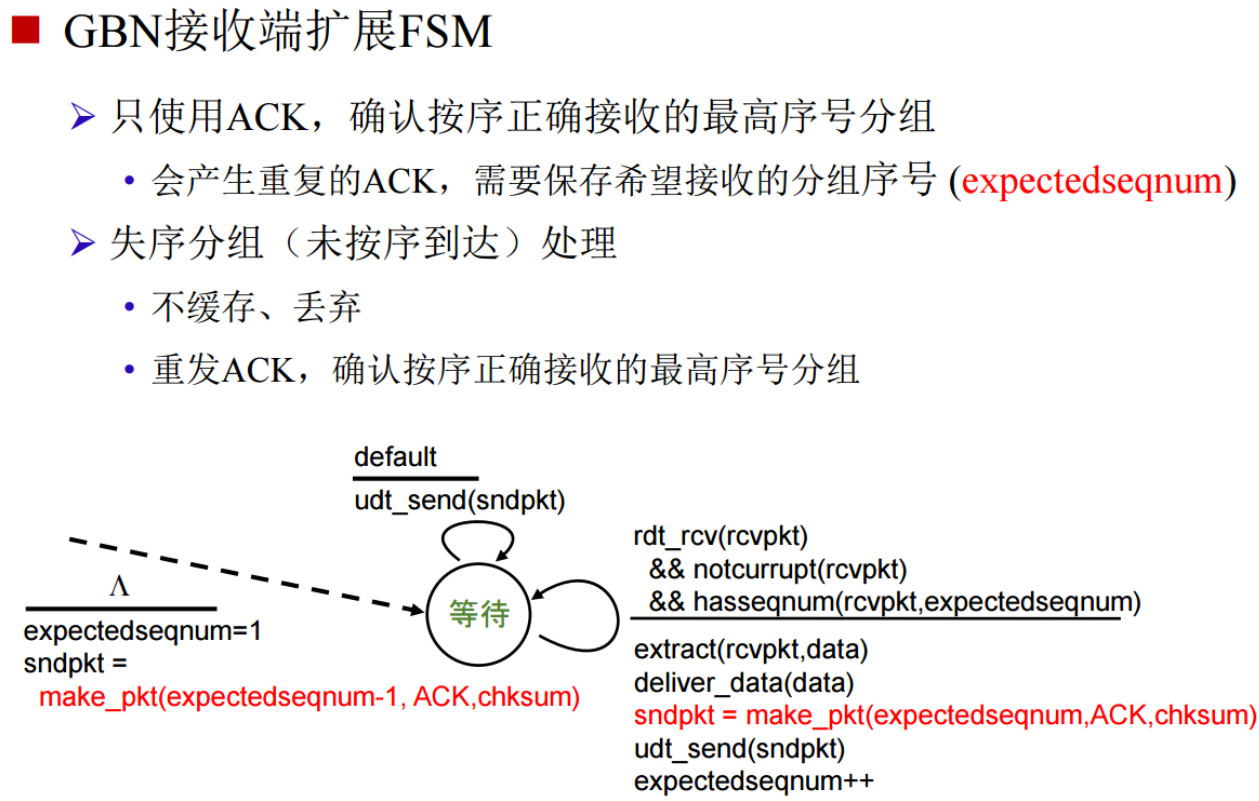


正常情况



滑动窗口机制包括以下几个关键概念：

- 发送窗口：发送方维护一组连续的允许发送的帧序号，称为发送窗口。
- 接收窗口：接收方设置接收窗口，用于控制哪些数据帧可以被接收，哪些不可以。只有当接收到的数据帧序号落在接收窗口内时，接收方才会接收该帧；否则，该帧将被丢弃。
- 窗口滑动机制：在发送端，每当收到一个有效的确认帧（ACK），发送窗口就会向前滑动一个帧的位置。如果发送窗口内没有可发送的帧，发送方将停止发送并进入等待状态，直到收到接收方的确认帧使窗口移动，窗口内有可发送的帧时，发送方才会继续发送。

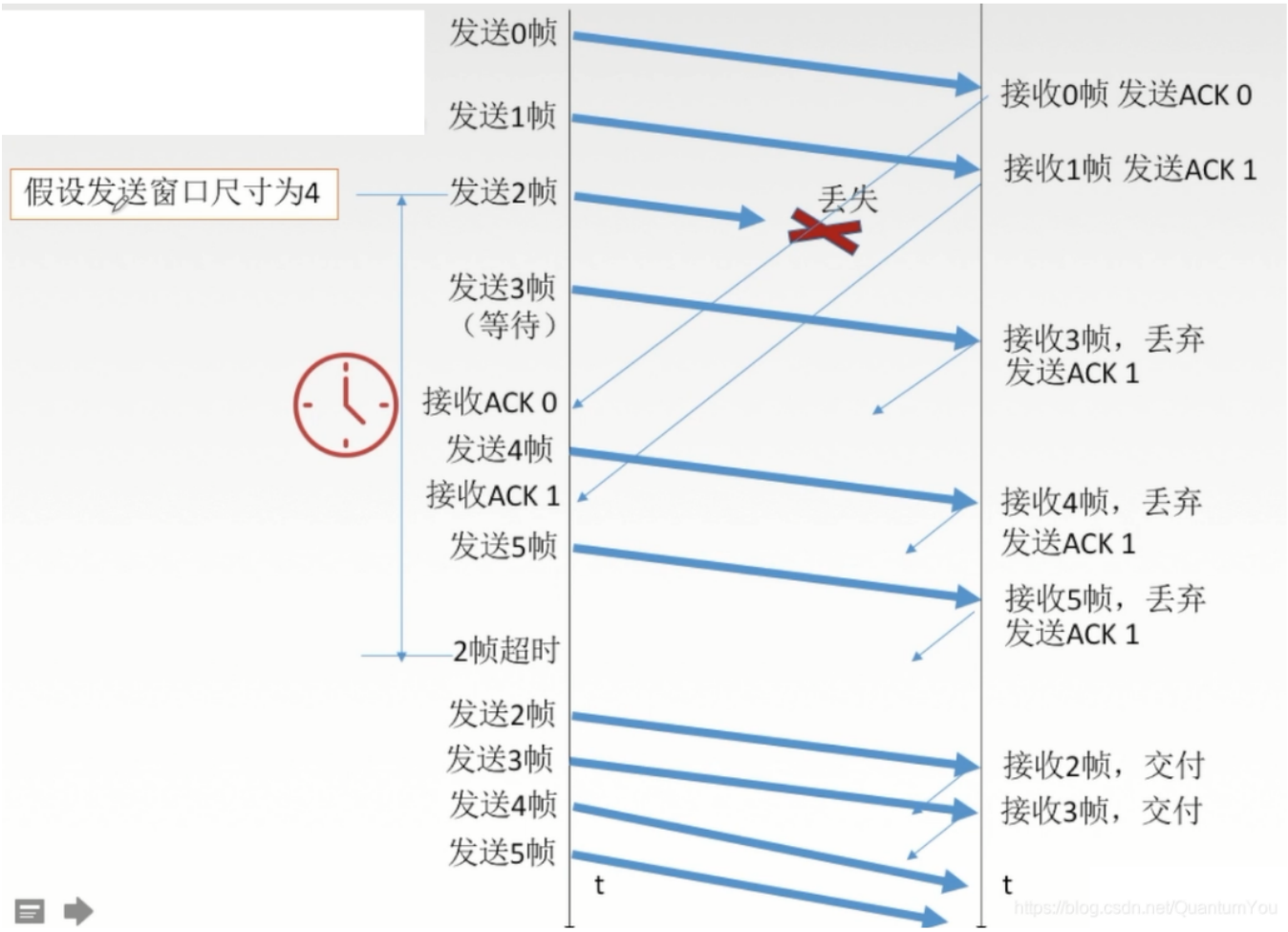


]

本实验采用的Go Back N滑动窗口协议，其特点是发送窗口大小大于1，而接收窗口大小为1。在这种协议下，发送方不需要在收到上一帧的ACK后才能发送下一帧。如果接收方检测到失序的帧，它会要求发送方重发最后一个正确帧之后的所有未确认帧；或者当发送方发送了N个帧后，如果发现前一个帧在计时器超时后仍未返回确认信息，则该帧被判定为出错或丢失，发送方将不得不重传该出错帧及之后的N个帧。

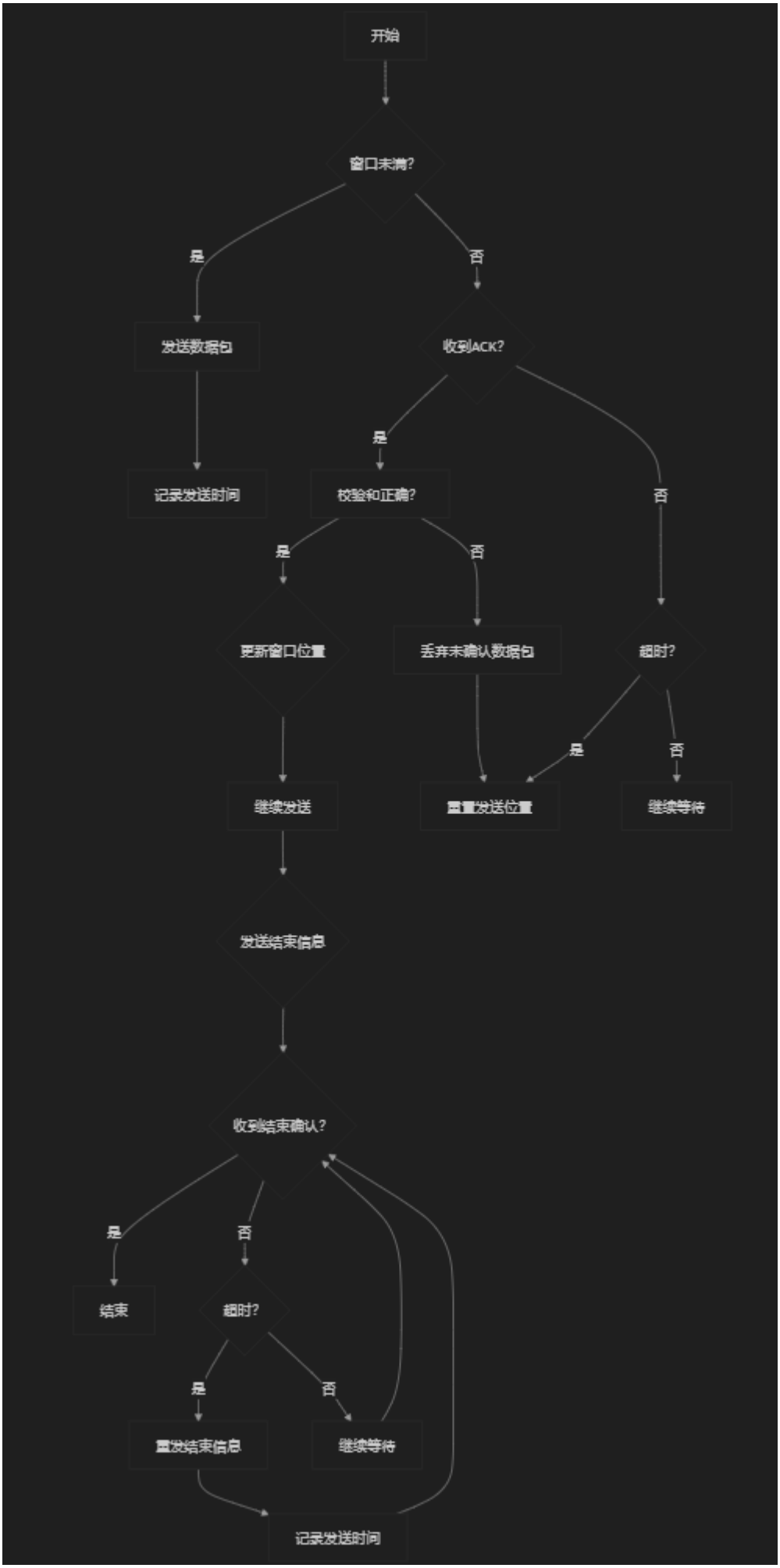
为了减少开销，Go Back N协议还规定，接收方不一定每收到一个正确的数据帧就必须立即发回一个确认，而是可以在连续收到多个正确的数据帧后，对最后一个数据帧发送确认信息，或者在有数据要发送时，对以前正确收到的帧进行捎带确认。

具体流程如下所示：



客户端发送数据

发送数据主要涉及到以下几个步骤：



```
void Send_Message(SOCKET &socketClient, SOCKADDR_IN &server_addr, int
&server_addr_len, char *message, int len)
{
    int package_num = len / BUFFER_SIZE + (len % BUFFER_SIZE != 0); // 数据包数量
    HEADER header;
    char *buffer = new char[BUFFER_SIZE + sizeof(header)];
    auto sendPacket = [&](int pos, bool isEnd)
    {
        int pack_len = (pos == package_num - 1) ? len - pos * BUFFER_SIZE :
BUFFER_SIZE;
        header.SEQ = unsigned char(pos % 256);
        header.datasize = pack_len;
        if (isEnd)
        {
            header.flag = END;
        }
        else
        {
            memcpy(buffer, message + pos * BUFFER_SIZE, pack_len);
        }
        header.sum = check_sum((unsigned short *)&header, sizeof(header) + (isEnd
? 0 : pack_len));
        memcpy(buffer, &header, sizeof(header));
        if (sendto(socketClient, buffer, sizeof(header) + (isEnd ? 0 : pack_len),
0, (sockaddr *)&server_addr, server_addr_len) == SOCKET_ERROR)
        {
            return false;
        }
        cout << "发送信息 " << pack_len << " bytes! SEQ:" << int(header.SEQ) <<
endl;
        return true;
    };

    // 已确认的最后一个数据包    当前窗口的下一个待发送数据包序列号
    int first_pos = -1, last_pos = 0;
    clock_t start;

    while (first_pos < package_num - 1)
    {
        if (last_pos != package_num && last_pos - first_pos < MAX_WINDOW)
        {
            if (!sendPacket(last_pos, false))
                break;
            start = clock();
            last_pos++;
        }
        unsigned long mode = 1;
        ioctlsocket(socketClient, FIONBIO, &mode);
        if (recvfrom(socketClient, buffer, sizeof(header), 0, (sockaddr
*)&server_addr, &server_addr_len) > 0)
        {
            memcpy(&header, buffer, sizeof(header));
```

```
            if (check_sum((unsigned short *)&header, sizeof(header)) == 0)
            {
                int temp = header.SEQ - first_pos % 256;
                if (temp > 0 || (header.SEQ < MAX_WINDOW && first_pos % 256 +
MAX_WINDOW >= 256))
                {
                    first_pos += temp;
                    cout << "发送已被确认! SEQ:" << int(header.SEQ) << endl;
                    cout << "窗口: " << first_pos << "~" << first_pos + MAX_WINDOW
<< endl;
                }
                else
                {
                    continue; // 忽略重复ACK
                }
            }
            else
            {
                last_pos = first_pos + 1;
                cout << "ERROR! 已丢弃未确认数据包" << endl;
            }
        }
        else if (clock() - start > MAX_WAIT_TIME)
        {
            last_pos = first_pos + 1;
            cout << "确认超时，开始重传...";
        }
        mode = 0;
        ioctlsocket(socketClient, FIONBIO, &mode);
    }
    // 发送结束信息
    while (true)
    {
        if (sendPacket(-1, true))
        {
            unsigned long mode = 1;
            ioctlsocket(socketClient, FIONBIO, &mode);
            if (recvfrom(socketClient, buffer, BUFFER_SIZE, 0, (sockaddr
*)&server_addr, &server_addr_len) > 0)
            {
                memcpy(&header, buffer, sizeof(header));
                if (check_sum((unsigned short *)&header, sizeof(header)) == 0 &&
header.flag == END)
                {
                    cout << "对方已成功接收文件!" << endl;
                    break;
                }
            }
            else if (clock() - start > MAX_WAIT_TIME)
            {
                cout << "发送超时! 开始重传..." << endl;
                start = clock();
            }
        }
```

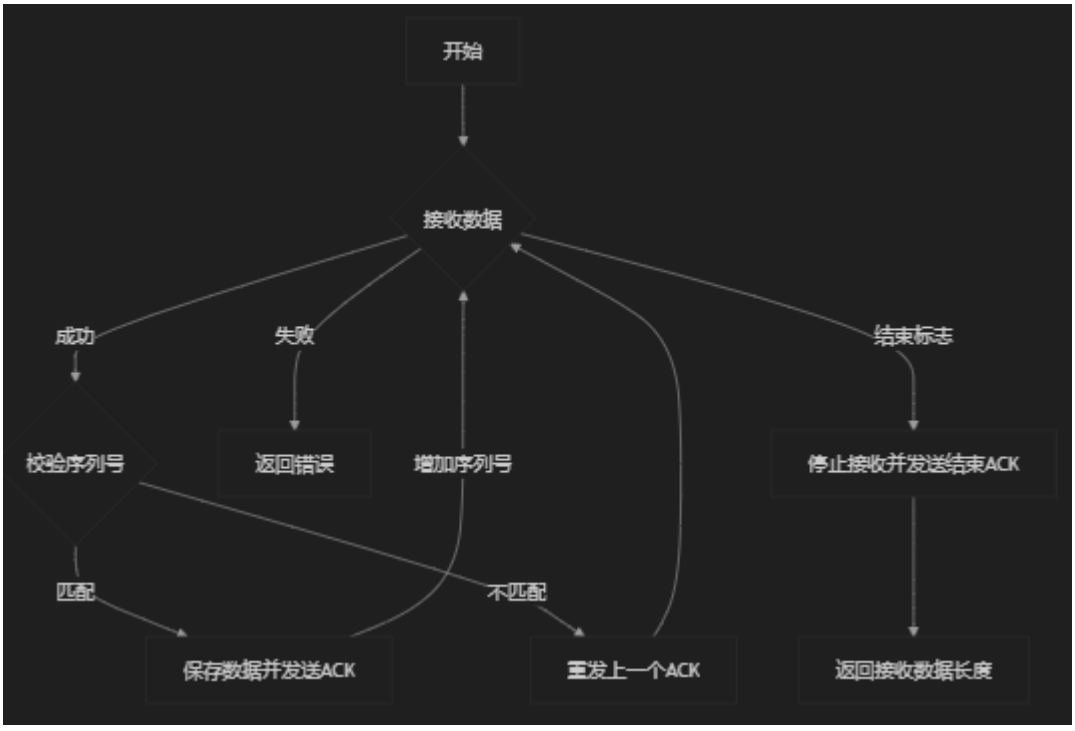
```
    }

    delete[] buffer; // 清理分配的内存
}
```

> 阻塞模式和非阻塞模式

- 阻塞和非阻塞是对操作请求者在等待返回结果时所处状态的描述。在阻塞模式下，当前线程会被挂起，直到操作请求的结果返回；而在非阻塞模式下，如果系统无法立即提供结果，线程不会被挂起，而是直接返回一个错误信息。因此，在非阻塞的情况下，调用者需要定期轮询以检查处理状态。
- 本质上，阻塞和非阻塞是对socket的不同处理方式，不会影响socket的连接及通信双方。通信的任一方都可以选择使用阻塞或非阻塞模式，例如，客户端采用阻塞模式，而服务器的`accept`可以设置为非阻塞模式，这都是可以的。
- 非阻塞是对阻塞方式的一种改进方案，通常能在大部分情况下提供更好的性能。虽然设置超时时间可以部分解决阻塞带来的问题，但它仍然会占用一定的时间，并且超时时间的设定往往较为复杂：若设置过短，可能导致操作过早中止；若设置过长则没有实际意义。因此，系统实现了非阻塞方式并结合异步编程，从而大幅提升了效率。

服务端接收数据

接收数据主要涉及到以下几个步骤：




```
int Receive_message(SOCKET &socketServer, SOCKADDR_IN &client_addr, int
&client_addr_len, char *message)
{
    HEADER header;
    char *buffer = new char[BUFFER_SIZE + sizeof(header)];
    int seq = 0; // 序列号
    int len = 0; // 已读取长度
    auto sendACK = [&](unsigned char ack_seq) {
        header.flag = ACK;
```

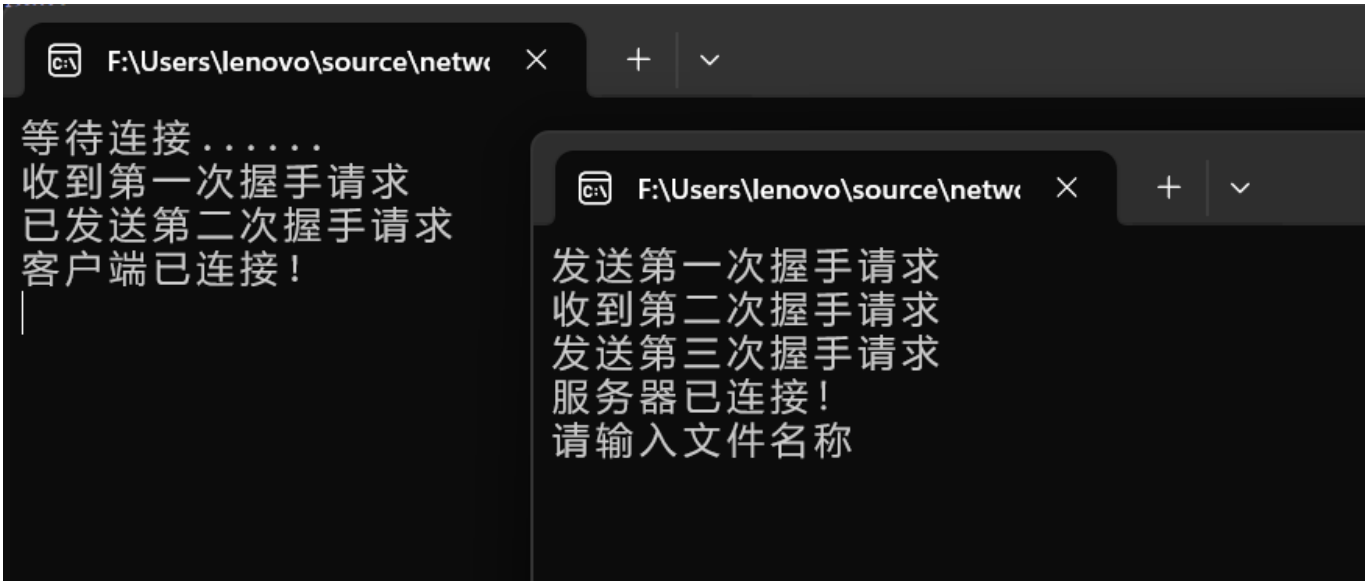
```
        header.datasize = 0;
        header.SEQ = ack_seq;
        header.sum = check_sum((unsigned short *)&header, sizeof(header));
        memcpy(buffer, &header, sizeof(header));
        if (sendto(socketServer, buffer, sizeof(header), 0, (sockaddr
*)&client_addr, client_addr_len) == SOCKET_ERROR)
        {
            return false;
        }
        cout << "已发送ACK! SEQ:" << (int)header.SEQ << endl;
        return true;
    };
    while (true)
    {
        int mess_len = recvfrom(socketServer, buffer, sizeof(header) +
BUFFER_SIZE, 0, (sockaddr *)&client_addr, &client_addr_len);
        if (mess_len <= 0) continue;
        memcpy(&header, buffer, sizeof(header));
        if (header.flag == 0)
        {
            if (header.SEQ == seq) // 序列号匹配
            {
                memcpy(message + len, buffer + sizeof(header), header.datasize);
                len += header.datasize;
                cout << "已接收数据 " << header.datasize << " bytes! SEQ: " <<
int(header.SEQ) << endl;
                if (!sendACK(seq))
                {
                    delete[] buffer;
                    return SOCKET_ERROR;
                }
                seq = (seq + 1) % 256; // 增加当前序列号并处理溢出
            }
            else // 序列号不匹配
            {
                cout << "已接收数据 " << header.datasize << " bytes! SEQ: " <<
int(header.SEQ) << endl;
                unsigned char last_ack = (seq - 1 + 256) % 256; // 确保不会出现负数
                if (!sendACK(last_ack))
                {
                    delete[] buffer;
                    return SOCKET_ERROR;
                }
                cout << "待接收序列号: " << seq << " 序列号无效，已重发ACK! SEQ:" <<
(int)last_ack << endl;
                continue;
            }
        }
        else if (header.flag == END && check_sum((unsigned short *)&header,
sizeof(header)) == 0)
        {
            cout << "文件已成功接收" << endl;
            break;
        }
```

```
    }
    // 发送结束确认
    header.flag = END;
    header.sum = check_sum((unsigned short *)&header, sizeof(header));
    memcpy(buffer, &header, sizeof(header));
    if (sendto(socketServer, buffer, sizeof(header), 0, (sockaddr *)&client_addr,
client_addr_len) == SOCKET_ERROR)
    {
        delete[] buffer;
        return SOCKET_ERROR;
    }
    delete[] buffer; // 清理分配的内存
    return len;
}
```

四、实验结果

(一)建立连接

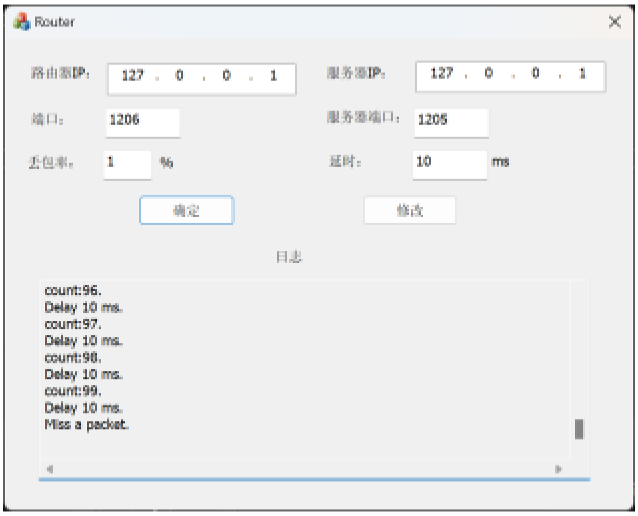
首先打开客户端和服务器端，握手建立连接如下图所示：



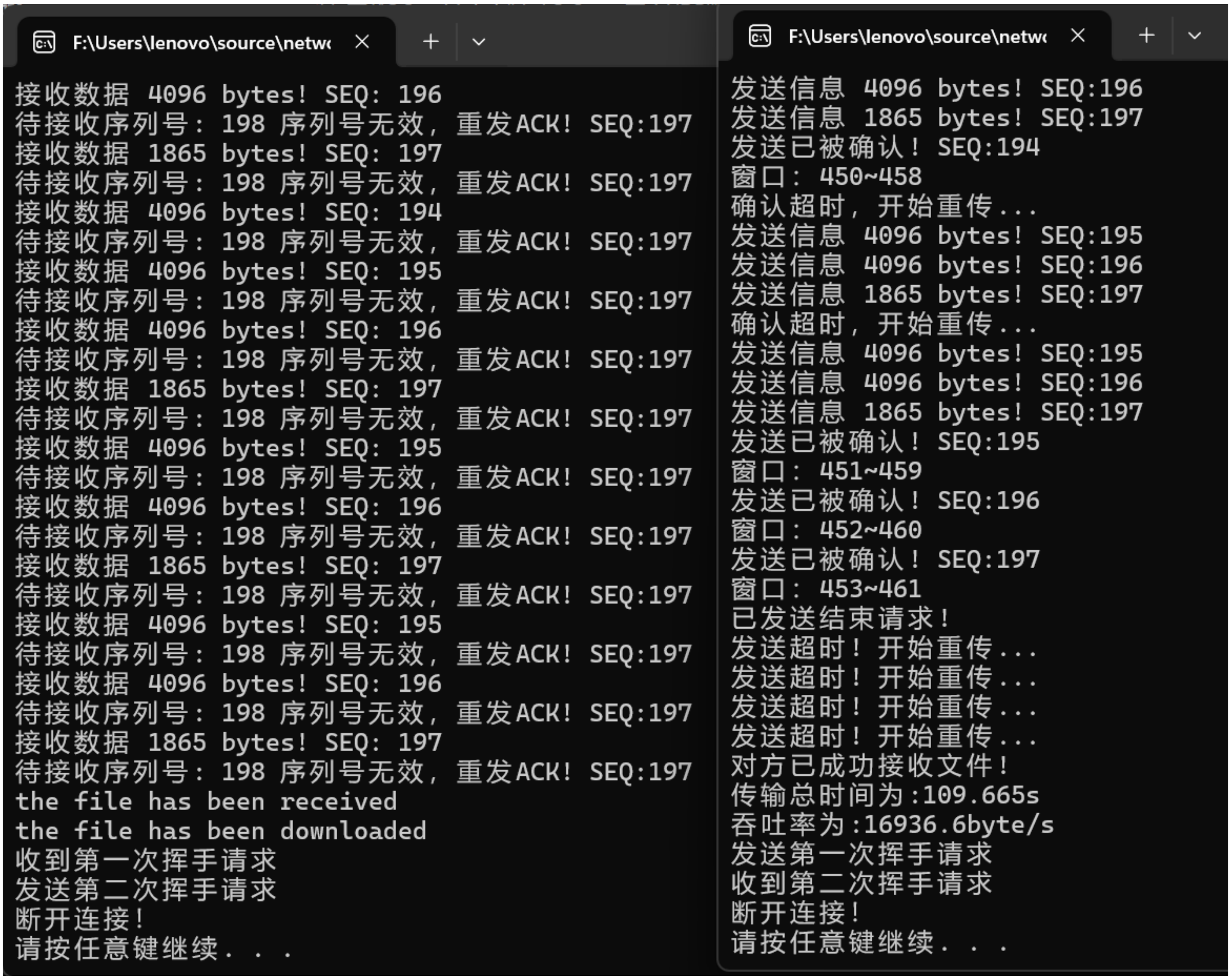
(二)数据传输

本次共需要传送四个文件，每个文件的传输及其传输后的总时间以及吞吐率如以下图片所示：

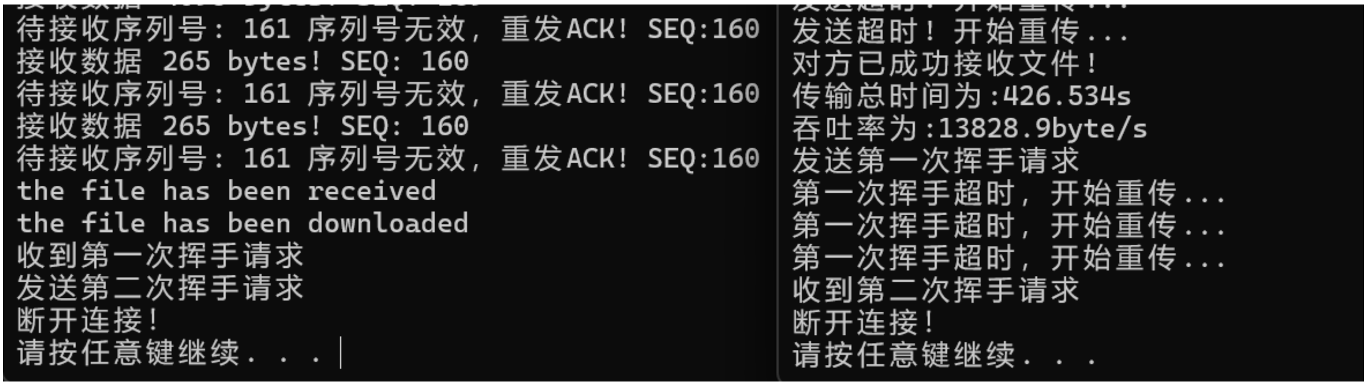
> 设置丢包


Router
路由器IP: 127 . 0 . 0 . 1
服务器IP: 127 . 0 . 0 . 1
端口: 1206
服务器端口: 1205
丢包率: 1 %
延时: 10 ms
确定
修改
日志
count:96.
Delay 10 ms.
count:97.
Delay 10 ms.
count:98.
Delay 10 ms.
count:99.
Delay 10 ms.
Miss a packet.

1.jpg

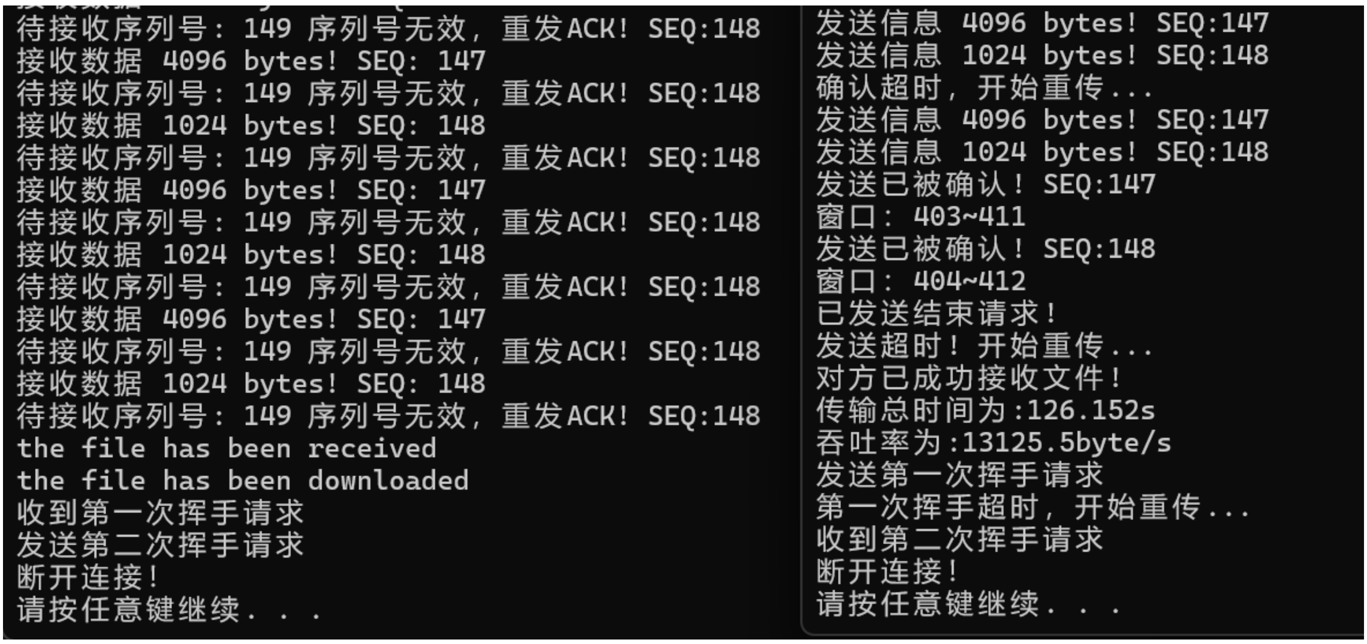

F:\Users\lenovo\source\netw
接收数据 4096 bytes! SEQ: 196
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 1865 bytes! SEQ: 197
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 4096 bytes! SEQ: 194
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 4096 bytes! SEQ: 195
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 4096 bytes! SEQ: 196
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 1865 bytes! SEQ: 197
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 4096 bytes! SEQ: 195
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 4096 bytes! SEQ: 196
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 1865 bytes! SEQ: 197
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 4096 bytes! SEQ: 195
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 4096 bytes! SEQ: 196
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
接收数据 1865 bytes! SEQ: 197
待接收序列号: 198 序列号无效, 重发ACK! SEQ:197
the file has been received
the file has been downloaded
收到第一次挥手请求
发送第二次挥手请求
断开连接!
请按任意键继续. . .
F:\Users\lenovo\source\netw
发送信息 4096 bytes! SEQ:196
发送信息 1865 bytes! SEQ:197
发送已被确认! SEQ:194
窗口: 450~458
确认超时, 开始重传...
发送信息 4096 bytes! SEQ:195
发送信息 4096 bytes! SEQ:196
发送信息 1865 bytes! SEQ:197
确认超时, 开始重传...
发送信息 4096 bytes! SEQ:195
发送信息 4096 bytes! SEQ:196
发送信息 1865 bytes! SEQ:197
发送已被确认! SEQ:195
窗口: 451~459
发送已被确认! SEQ:196
窗口: 452~460
发送已被确认! SEQ:197
窗口: 453~461
已发送结束请求!
发送超时! 开始重传...
发送超时! 开始重传...
发送超时! 开始重传...
发送超时! 开始重传...
对方已成功接收文件!
传输总时间为:109.665s
吞吐率为:16936.6byte/s
发送第一次挥手请求
收到第二次挥手请求
断开连接!
请按任意键继续. . .

2.jpg



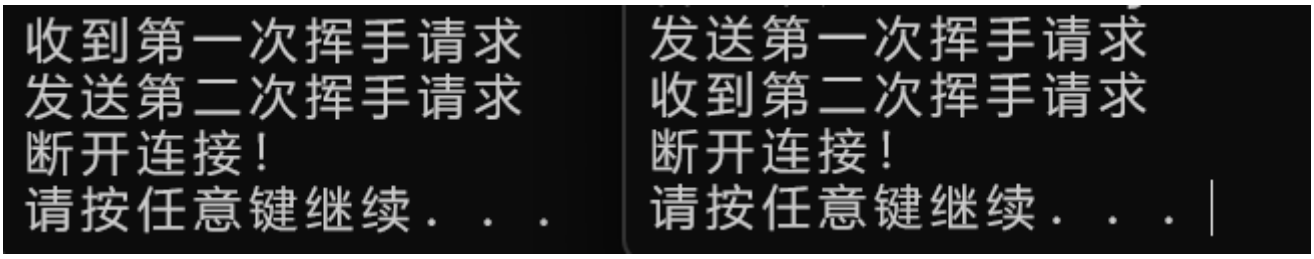
> 3.jpg

> helloworld.txt



(三)断开连接

- 挥手断开连接及文件的存储如下图所示：



(四)结果展示

接收到的文件：

名称	修改日期	类型	大小
x64	2024/12/3 0:14	文件夹	
1.jpg	2024/12/6 14:58	JPG 图片文件	1,814 KB
2.jpg	2024/12/6 14:55	JPG 图片文件	5,761 KB
3.jpg	2024/12/6 15:23	JPG 图片文件	11,689 KB
helloworld.txt	2024/12/6 15:34	文本文档	1,617 KB
server.cpp	2024/12/6 14:44	C++ Source	10 KB
server.vcxproj	2024/12/3 0:14	VC++ Project	7 KB
server.vcxproj.filters	2024/12/3 0:14	VC++ Project Filter...	1 KB
server.vcxproj.user	2024/12/3 0:14	Per-User Project O...	1 KB

可以看到四个文件都与原文件信息完全相同！

五、遇到的问题与总结

(1)问题

> 本次实验主要遇到以下几个问题：

- 最初的程序在大部分窗口大小下能够正确传输给定的测试文件，但在某些窗口大小下，传输2.jpg和3.jpg时，传输到第770数据包左右，会持续不断地重传，导致传输失败。
- 没有采用多线程，传输时间比较长。下次实验改进为多线程传输，提升传输效率。

(2)总结

在本次实验中，在实验3-1的基础上引入了基于滑动窗口的流量控制策略以支持累积确认。这一过程深刻地展示了如何在不可靠的传输层协议之上构建可靠的通信服务。面对并解决了诸如包丢失、乱序到达以及重复包等问题，同时确保了传输效率和数据完整性。通过对固定窗口大小的管理，有效地控制了发送方的数据流速，防止了接收方的缓冲区溢出。此实验不仅强化了我对网络编程原理的理解，也提升了实际问题解决能力，尤其是在实现复杂网络协议方面。最终，成功完成了给定测试文件的无误传输，验证了所设计系统的稳定性和可靠性。