Lab3-3——基于UDP服务设计可靠传输协议并编程实现

学号：2013921
姓名：周延霖
专业：信息安全

一、实验要求

在实验3-2的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

二、实验设计

(一)原理探究

1.三次握手

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。

2.四次挥手

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。

3.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。

4.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。

5.流水线协议

流水线协议允许在ACK未返回之前允许发送多个分组，主要分为以下两种方法：

- Go-Back-N（GBN）返回N
    - 窗口大小为N，最多允许N个分组未确认
    - ACK(n),则表示确认从开始到 n（包含n）的序列号全部正确接收
    - 空中在传的分组设置一个Timer计时器，处理超时，如果收到了timeout(n)事件，那么会重传的是 n 以及 n 以后的所有分组（尽管后面的可能已经收到了，这就是回退，回退到n开始传，GBN）
- Selective Repeat（SR）选择重传
    - GBN缺陷，累积确认机制导致回退到N，重复传了很多。SR 解决了这个缺陷
    - 对每个分组分别确认，不再只接收期望的，接到不期望的，就先缓存（设置缓存机制），接到期望的才交付上层
    - 发送方只需要重传那些没收到ACK的分组
    - 产生了接收方窗口（GBN只有发送方窗口），用来缓存，现在有两窗口了

本次实验采用的是Go-Back-N

> 6.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。

> 7.滑动窗口

滑动窗口(Sliding window)是一种流量控制技术。如果网络通信中，通信双方不会考虑网络的拥挤情况直接发送数据，由于大家不知道网络拥塞状况，同时发送数据，则会导致中间节点阻塞掉包，谁也发不了数据， 所以就有了滑动窗口机制来解决此问题。

TCP中采用滑动窗口来进行传输控制，滑动窗口的大小意味着接收方还有多大的缓冲区可以用于接收数据。发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为0时，发送方一般不能再发送数据报，但有两种情况除外，一种情况是可以发送紧急数据，例如，允许用户终止在远端机上的运行进程。另一种情况是发送方可以发送一个1字节的数据报来通知接收方重新声明它希望接收的下一字节及发送方的滑动窗口大小。

> 8.拥塞控制

在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络性能就要变坏，这种情况就叫做网络拥塞。在计算机网络中数位链路容量（即带宽）、交换结点中的缓存和处理机等，都是网络的资源。若出现拥塞而不进行控制，整个网络的吞吐量将随输入负荷的增大而下降。

- 目标：既不造成网络严重拥塞，又能更快地传输数据
- 带宽探测：接收到ACK，高传输速率；发生丢失事件，降低传输速率
- ACK返回：说明网络并未拥塞，可以继续提高发送速率
- 丢失事件：假设所有丢失是由于拥塞造成的，降低发送速率

> 9.控制算法

Tahoe：

Tahoe算法是TCP的早期版本。除了具备TCP的基本架构和功能外，引入了慢启动、拥塞避免以及快速重传机制。 在该算法中，快速重传机制策略如下：

- ssthresh设置为拥塞窗口的1/2
- 拥塞窗口大小设置为1
- 重新进入慢启动阶段

Reno：

Reno与Tahoe相比，增加了快速恢复阶段，也就是说，完成快速重传后，进入了拥塞避免阶段而不是慢启动阶段。 Reno 在快速重传阶段，重新发送数据之后：

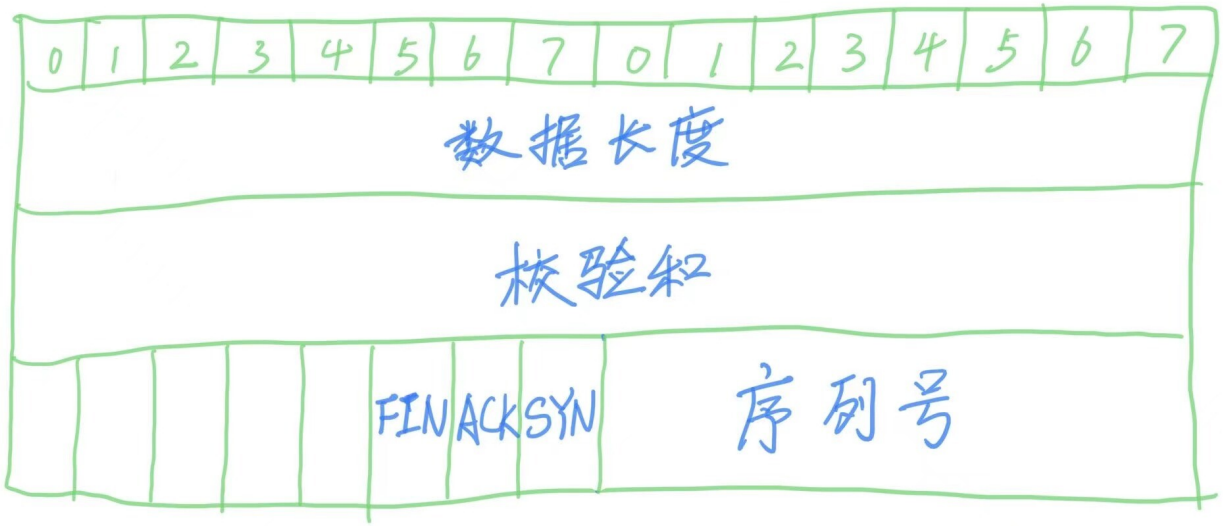
- ssthresh设置为拥塞窗口的1/2
- 拥塞窗口设置为之前的1/2
- 进入快速恢复阶段
- 在快速恢复阶段，每收到重复的ACK，则cwnd加1；收到非重复ACK时，置cwnd = ssthresh，转入拥塞避免阶段；
- 如果发生超时重传，则置ssthresh为当前cwnd的一半，cwnd = 1，重新进入慢启动阶段。

Reno快速恢复阶段退出条件：收到非重复ACK。(本次个人也是采用的reno算法)

(二)协议设计

1.报文格式

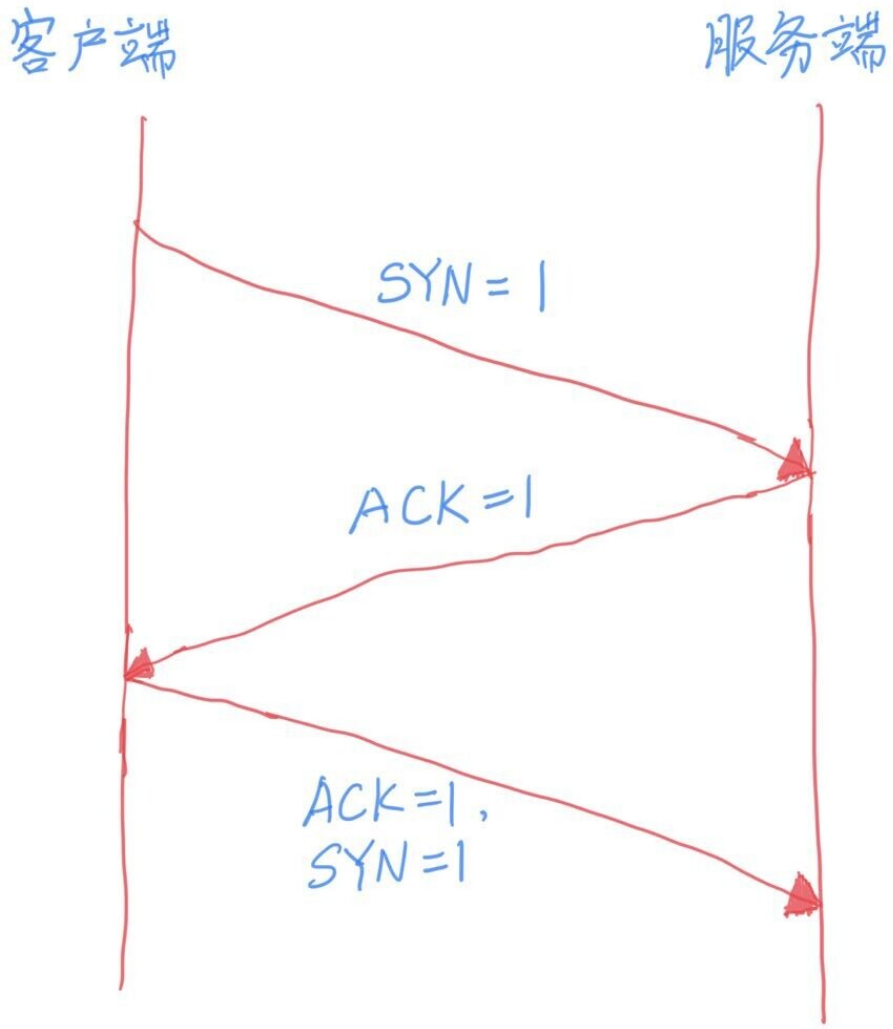
所设计的报文格式如下图所示：



- 报文头总长度为48位
- 前16位为数据长度，用于记录数据区的长度大小
- 17-32位为校验和，用于检验传输数据的正确性
- 33-40位为标志位，在本次实验中只使用低3位，分别为FIN，ACK，SYN
- 41-48位为传输的数据包的序列号，用于区分顺序，0-255循环使用

2.三次握手建立连接

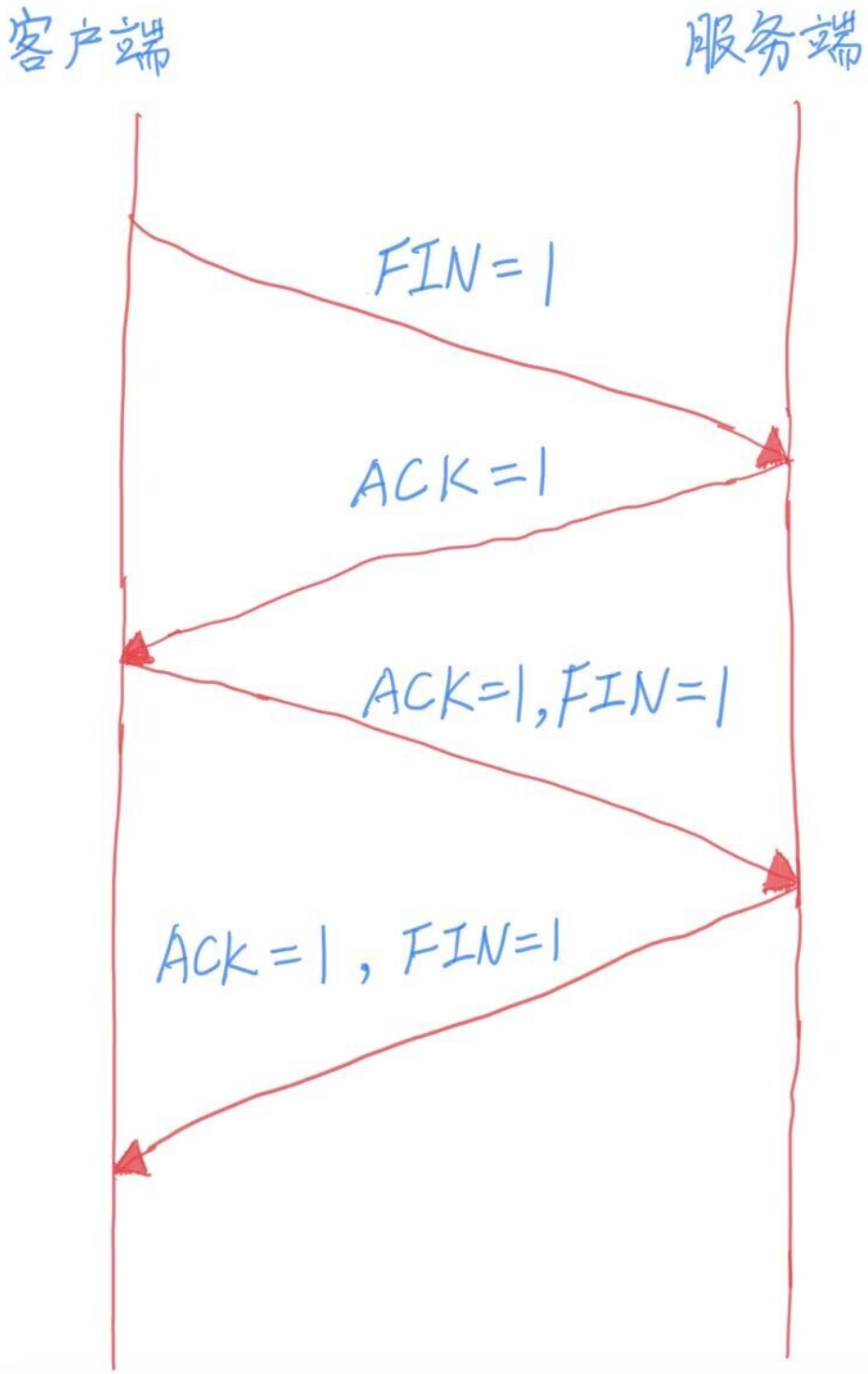
- 根据三次握手的原理所设计的流程图如下图所示：



1. 首先，客户端向服务端发送数据报，其中SYN=1，ACK=0，FIN=0（第一次握手）
2. 服务端接受到数据报后，向客户端发送SYN=0，ACK=1，FIN=0（第二次握手）
3. 客户端再次接收到数据报后，向服务端发送SYN=1，ACK=1，FIN=0（第三次握手）
4. 服务端接收到数据报后，连接成功建立，三次握手完成，可以进行数据传输

3.四次挥手断开连接

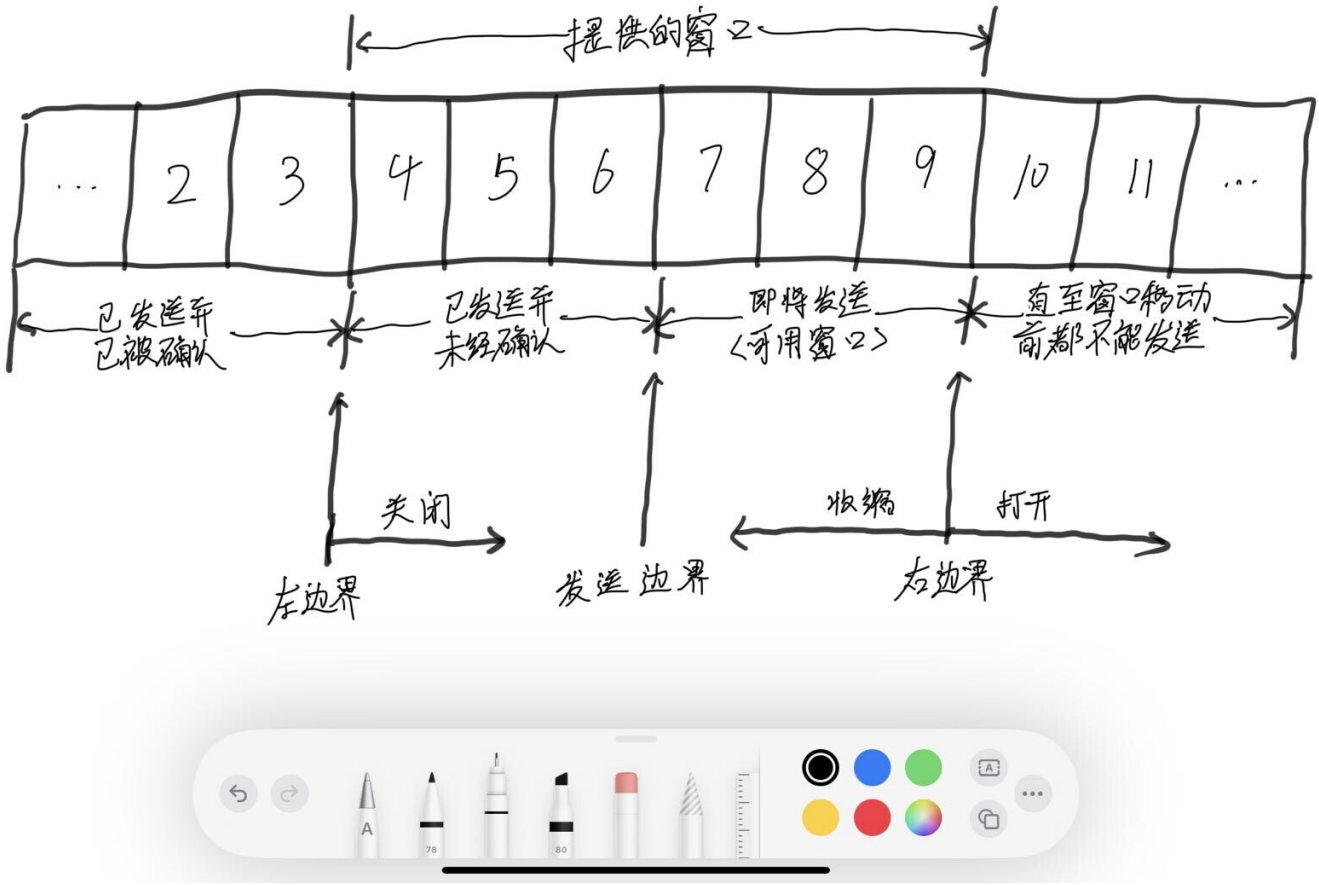
- 根据四次挥手的原理所设计的流程图如下图所示：



1. 首先，客户端向服务端发送数据报，其中SYN=0，ACK=0，FIN=1（第一次挥手）
2. 服务端接受到客户端发来的数据报后，向客户端发送SYN=0，ACK=1，FIN=0（第二次挥手）
3. 客户端再次接收到数据报后，向服务端发送SYN=0，ACK=1，FIN=1（第三次挥手）
4. 服务端接收到数据报后，向客户端发送SYN=0，ACK=1，FIN=1（第四次挥手）
5. 客户端接收到数据报后，四次挥手完成，成功断开连接

4.滑动窗口

关于滑动窗口的设计如下：

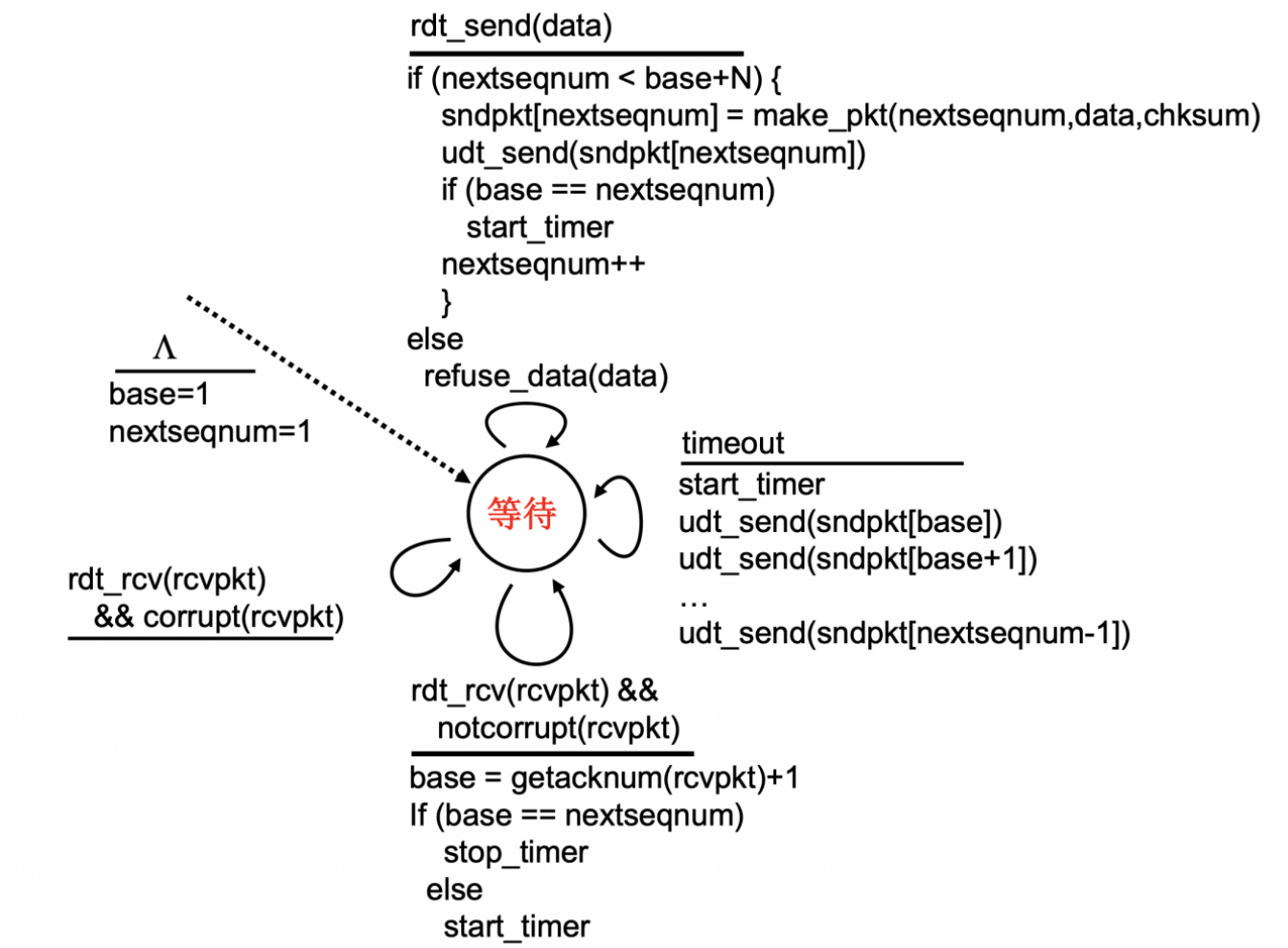


- 窗口分为左边界、发送边界和右边界
- 窗口大小固定
- 窗口左边界左侧为已经发送并得到确认的数据
- 左边界到发送边界的数据为已发送但未得到确认的数据
- 发送边界到右边界为等待发送的数据
- 右边界右侧为不可发送的数据

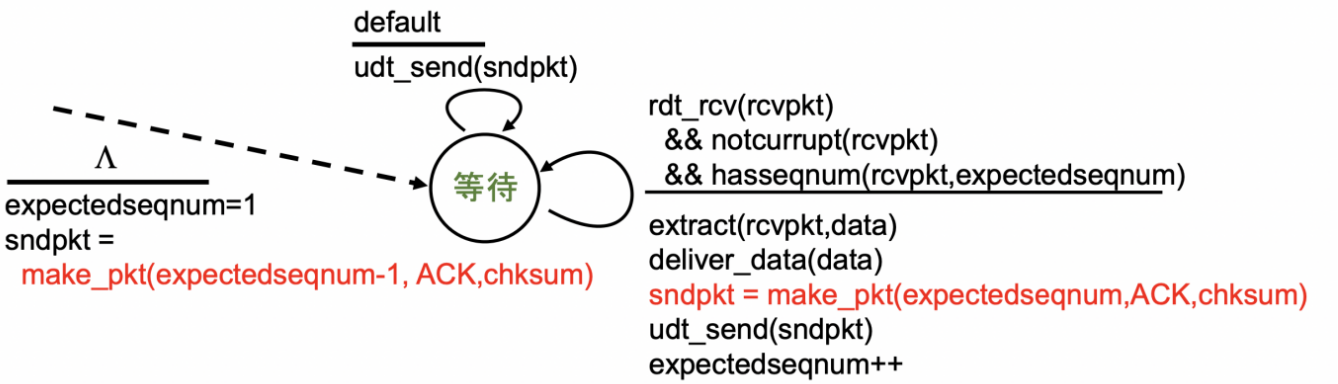
5.数据传输

发送端和接收端的接收机均采用GBN的设计原则，由于在一开始需要了解的是发送过程，所以不用实现双向传送，所以只需要一边是接收端，另一边是发送端即可

- 发送端的有限状态机如下图所示：



- 接收端的有限状态机如下图所示：



- 由于数据的传输一般都会大于最大每次传输数据的长度，所以在数据传输时，将需要传输的文件分为多个数据报进行分段传输，每个包的内容为`数据头 + 数据`。
- 在传输时，由于采用的是累计确认，所以无需接受到上一个发送包序号的`ACK = 1`才能发送下一个数据包，可以继续发送直到窗口大小数量的数据包
- 接收端接收到了一个数据包，先要进行校验，如果检查无误，则向发送放返回该序列号的`ACK = 1`
- 在一定时间内，如果没有收到某一已经传输的报文的正确的ACK，则将窗口中所有位于改数据包后已经传输但未得到确认的数据包丢弃，并从该数据包开始进行重传
- 如果再次接收到了已经确认的报文的ACK，则忽略此数据报
- 如果接收端收到了重复的包裹，则将其中一个丢弃，但仍需要向发送方发送该序列号的`ACK = 1`
- 最后，发送方需要向接收端发送一个FIN=1， ACK=1，SYN=1的包裹，表示文件传输结束

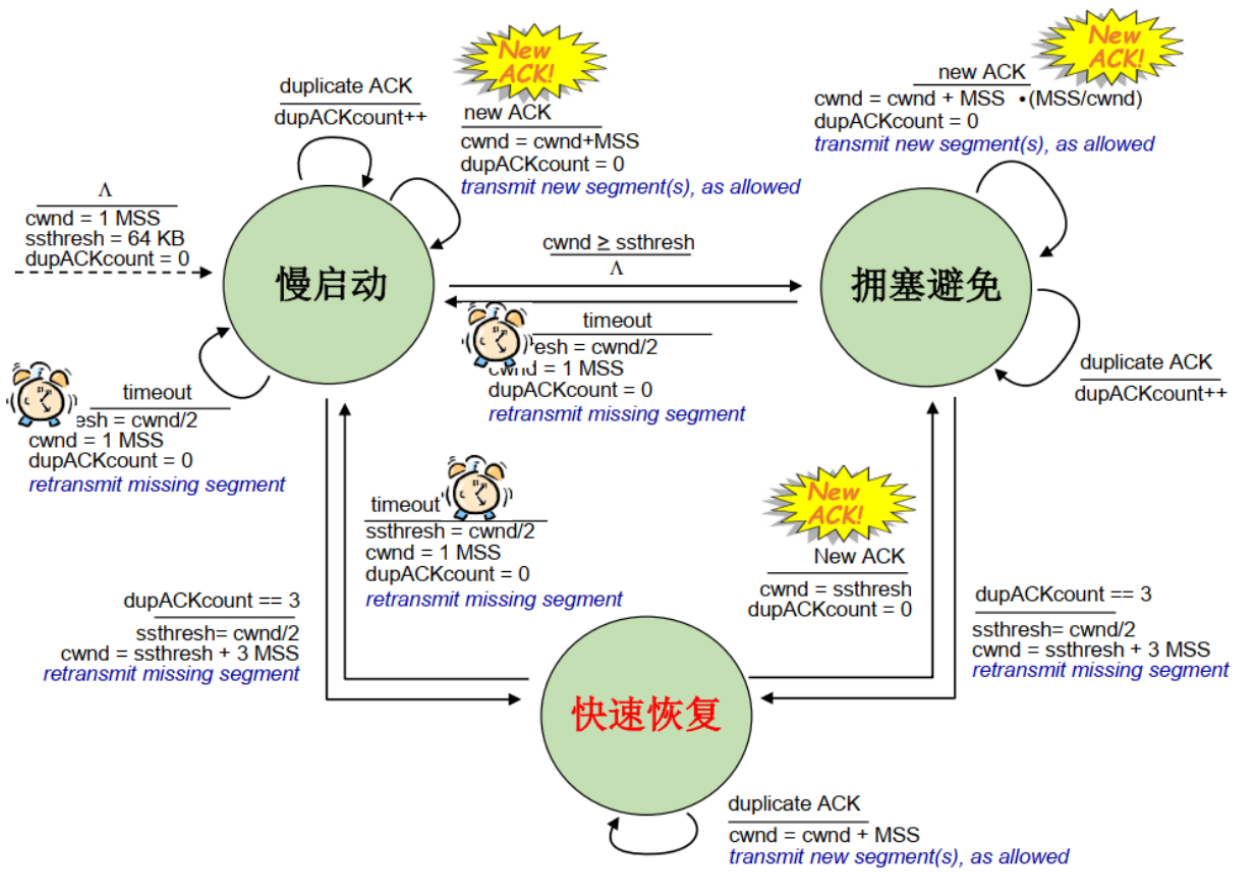
- 接收端收到结束信号后，需要向发送方返回一个ACK=1，表示收到文件传输结束的信号

丢包设计

本次实验利用老师发的路由器小程序进行丢包和延时，分别将路由的端口绑定在92将服务器的端口绑定在90，然后设置丢包和延时检验程序的正确性。

reno算法

整个算法大致如下图所示：



阶段介绍：

- 慢启动阶段：初始拥塞窗口为`cwnd = 1`，每收到一个新的ACK，cwnd自增1。当连接初始建立或超时，则进入慢启动阶段
- 拥塞避免阶段：设置阈值ssthresh，当拥塞窗口`cwnd >= ssthresh`时，则从慢启动阶段进入拥塞避免阶段。拥塞避免阶段中，每轮RTT，cwnd增加1
- 快速恢复阶段：当接收到三次冗余ACK时，进入快速恢复阶段。在快速恢复阶段中，每收到一个冗余ACK，cwnd增1。如果出现超时，则进入慢启动阶段；如果接收到新的ACK，进入拥塞避免阶段

事件处理：

- 超时：当超时发生时，ssthresh设置为`cwnd / 2`，cwnd设为1，并进入慢启动阶段
- 新ACK：如果处于快速恢复阶段，收到新ACK后会进入拥塞避免阶段，并设置`cwnd = ssthresh`。如果当前处于其余两个阶段，则cwnd进行相应的增加
- 三次冗余ACK：收到三次冗余ACK，ssthresh设为`cwnd / 2`，cwnd设为`ssthresh + 3`，并进入快速恢复阶段

三、设计实现

> 本次实验基本思想除了拥塞控制外与前两次基本类似，比如三次握手、四次挥手等，所以重复的部分就不再赘述代码，只展示更改的重要部分：

(一)方便实现reno算法的结构体的定义

首先是zyl_reno结构体的定义，采用reno的拥塞控制算法：

- 此结构体里的变量包括拥塞窗口的大小，阈值大小，重复ack的个数，以及到拥塞避免每窗口大小增加1的记录数值和三个状态的定义
- 结构体里的函数首先是接收到新的数据报的函数的处理：
  - 如果是慢启动状态，则窗口大小加一，然后根据窗口是否大于阈值判断是否进入拥塞避免阶段
  - 如果是拥塞避免阶段，则每隔窗口大小增加一个大小
  - 如果是快速恢复阶段则让拥塞窗口为阈值并进入拥塞避免阶段
- 接下来是超时的处理：
  - 不论现在在哪一个状态，都必须进入慢启动阶段并且让阈值为当前窗口的一半并让窗口为1
- 然后是接收到重复数据报的处理，当重复的数据报大于3的时候：
  - 如果现在是慢启动或者拥塞避免阶段则让阈值为当前窗口的一半并让窗口为阈值加3，进入快速恢复阶段
  - 如果当前是快速恢复阶段，则窗口大小增加1

```
class zyl_reno
{
private:
    u_int cwnd;
    u_int ssthresh;
    u_int dup_ack;
    u_int num_to_add_one;
    enum
    {
        SLOW_START,
        CONGESTION_AVOID,
        QUICK_RECOVER
    } state;
public:
    zyl_reno()
    {
        cwnd = 1;
        ssthresh = INITSSTHRESH;
        dup_ack = 0;
        state = SLOW_START;
    };
    u_int get_cwnd()
    {
        return cwnd;
    }
    u_int get_dup_ack()
    {
        return dup_ack;
```

```
    }
    void rec_new() {
        dup_ack = 0;
        switch (state)
        {
            case SLOW_START:
            {
                cwnd += 1;
                if (cwnd >= ssthresh)
                {
                    num_to_add_one = cwnd;
                    state = CONGESTION_AVOID;
                }
            }break;
            case CONGESTION_AVOID:
            {
                num_to_add_one -= 1;
                if (num_to_add_one <= 0)
                {
                    cwnd += 1;
                    num_to_add_one = cwnd;
                }
            }break;
            case QUICK_RECOVER:
            {
                cwnd = ssthresh;
                state = CONGESTION_AVOID;
                num_to_add_one = cwnd;
            }; break;
        }
    }
    void OutOfTime()
    {
        dup_ack = 0;
        ssthresh = cwnd / 2;
        cwnd = 1;
        state = SLOW_START;
    }
    void DupAck() {
        dup_ack += 1;
        if (dup_ack < 3)
        {
            return;
        }
        switch (state)
        {
            case SLOW_START:
            case CONGESTION_AVOID:
            {
                ssthresh = cwnd / 2;
                cwnd = ssthresh + 3;
                state = QUICK_RECOVER;
            }break;
            case QUICK_RECOVER:
```

```
            {
                cwnd += 1;
            }break;
        }
    }
};
```

(二)接收线程的定义

1. 每次判断是否完成所有的发送确认，如果完成则结束处理并返回
2. 如果还没有完成所有的发送确认，则判断是否超时
3. 如果超时则利用go-back-n重传数据报，如果并不超时且校验和正确摒弃是ACK报文，则判断是新数据报还是重复数据报
4. 如果是新数据报则直接让结构体处理即可
5. 如果是重复数据报则发送丢失数据报，然后更改缓冲区，判断是否还有未确认的报文来看是否重新计时

```
void rec_thread(SOCKET* server, SOCKADDR_IN* server_addr)
{
    // 开启非阻塞模式
    u_long mode = 1;
    ioctlsocket(*server, FIONBIO, &mode);
    char* re_buf = new char[sizeof(Header)];
    Header* header;
    SOCKADDR_IN s_ad;
    int s_ad_len = sizeof(SOCKADDR_IN);
    while (true)
    {
        if (finish_send)
        {
            // 改为阻塞模式
            mode = 0;
            ioctlsocket(*server, FIONBIO, &mode);
            delete[]re_buf;
            return;
        }
        while (recvfrom(*server, re_buf, sizeof(Header), 0,
(sockaddr*)&s_ad, &s_ad_len) <= -1)
        {
            if (finish_send) {
                // 改为阻塞模式
                mode = 0;
                ioctlsocket(*server, FIONBIO, &mode);
                delete[]re_buf;
                return;
            }
            if (zyl_timer.out_of_time())
            {
                // 超时处理
                real_zyl_reno.OutOfTime();
```

```
                for (auto packet : GoBackN_buf)
                {
                    sendto(*server, (char*)packet, sizeof(Header) +
packet->header_pa.length, 0, (sockaddr*)server_addr, sizeof(SOCKADDR_IN));
                    cout_lock.lock();
                    cout << "数据报重传 其首部为: seq:" << packet-
>header_pa.seq << "; ack:" << packet->header_pa.ack << "; flag:" << packet-
>header_pa.flag << "; 校验和:" << packet->header_pa.chsum << "; len:" <<
packet->header_pa.length << endl;
                    cout_lock.unlock();
                }
                // 重新计时
                zyl_timer.start_timer();
            }
        }
        header = (Header*)re_buf;
        int chksum = chsum(re_buf, sizeof(Header));
        if (chksum != 0)
        {
            continue;
        }
        if (header->flag == RST)
        {
            cout << "连接异常 退出程序" << endl;
            exit(-1);
        }
        else if (header->flag == ACK)
        {
            if (header->ack >= base)
            {
                // 处理新数据报
                real_zyl_reno.rec_new();
            }
            else {
                // 处理重复
                real_zyl_reno.DupAck();
                if (real_zyl_reno.get_dup_ack() == 3)
                {
                    Packet* packet = GoBackN_buf[0];
                    sendto(*server, (char*)packet, sizeof(Header) +
packet->header_pa.length, 0, (sockaddr*)server_addr, sizeof(SOCKADDR_IN));
                    // 保证输出时的正确性
                    cout_lock.lock();
                    cout << "已有三个重复ACK 进行快速重传 其首部为: seq:" <<
packet->header_pa.seq << "; ack:" << packet->header_pa.ack << "; flag:" <<
packet->header_pa.flag << "; 校验和:" << packet->header_pa.chsum << "; len:"
<< packet->header_pa.length << endl;
                    cout_lock.unlock();
                }
            }
            // 定义接收到的数据
            int re_ack_num = header->ack + 1 - base;
            for (int i = 0; i < re_ack_num; i++)
            {
```

```
                // 防止go-back-n的缓冲区出错
                buf_lock.lock();
                if (GoBackN_buf.size() <= 0)
                {
                    break;
                }
                delete GoBackN_buf[0];
                // 更改缓冲区
                GoBackN_buf.erase(GoBackN_buf.begin());
                buf_lock.unlock();
            }
            base = header->ack + 1;
            cout_lock.lock();
            cout << "成功接收一个数据包 其首部为: seq:" << header->seq << "; 
ack:" << header->ack << "; flag:" << header->flag << "; 校验和:" << header-
>chsum << "; len:" << header->length << "; 拥塞窗口大小为:" << 
real_zyl_reno.get_cwnd() << endl;
            cout_lock.unlock();
        }
        if (base != nextseqnum)
        {
            zyl_timer.start_timer();
        }
        else
        {
            zyl_timer.stop_timer();
        }
    }
}
```

(三)发送单个数据报的定义

- 接下来是发送单个数据报，如果小于窗口大小则发送并加入缓冲区，具体代码如下：

```
void send_one(SOCKET* server, SOCKADDR_IN* server_addr, char* msg, int 
len, bool last = false)
{
    assert(len <= MSS);
    // 检查是否可以发送
    while ((u_short)(nextseqnum - base) >= WINDOWS || (u_short)(nextseqnum 
- base) >= real_zyl_reno.get_cwnd())
    {
        continue;
    }
    u_int zyl_win = real_zyl_reno.get_cwnd();
    Packet* zyl_pa = new Packet;
    zyl_pa->header_pa.set_header(nextseqnum, 0, last ? LAS : 0, 0, len);
    memcpy(zyl_pa->data, msg, len);
    u_short chSum = chsum((char*)zyl_pa, sizeof(Header) + len);
    zyl_pa->header_pa.chsum = chSum;
```

```
    buf_lock.lock();
    GoBackN_buf.push_back(zyl_pa);
    buf_lock.unlock();
    sendto(*server, (char*)zyl_pa, len + sizeof(Header), 0,
(sockaddr*)server_addr, sizeof(SOCKADDR_IN));

    cout_lock.lock();
    if (zyl_win <= WINDOWS)
    {
        cout << "发送一个数据包 其首部为: seq:" << zyl_pa->header_pa.seq <<
";ack:" << zyl_pa->header_pa.ack << ";flag:" << zyl_pa->header_pa.flag <<
";校验和:" << zyl_pa->header_pa.chsum << "; len:" << zyl_pa-
>header_pa.length << ";剩余发送窗口大小为:" << zyl_win - (nextseqnum - base) -
1 << endl;
    }
    else
    {
        cout << "发送一个数据包 其首部为: seq:" << zyl_pa->header_pa.seq <<
";ack:" << zyl_pa->header_pa.ack << ";flag:" << zyl_pa->header_pa.flag <<
";校验和:" << zyl_pa->header_pa.chsum << "; len:" << zyl_pa-
>header_pa.length << ";剩余发送窗口大小为:" << WINDOWS - (nextseqnum - base) -
1 << endl;
    }
    cout_lock.unlock();

    if (base == nextseqnum)
    {
        zyl_timer.start_timer();
    }
    nextseqnum += 1;
}
```

(四)发送文件的定义

- 然后发送文件中先开启接收线程在发送数据报，剩余部分与之前类似，具体代码如下：

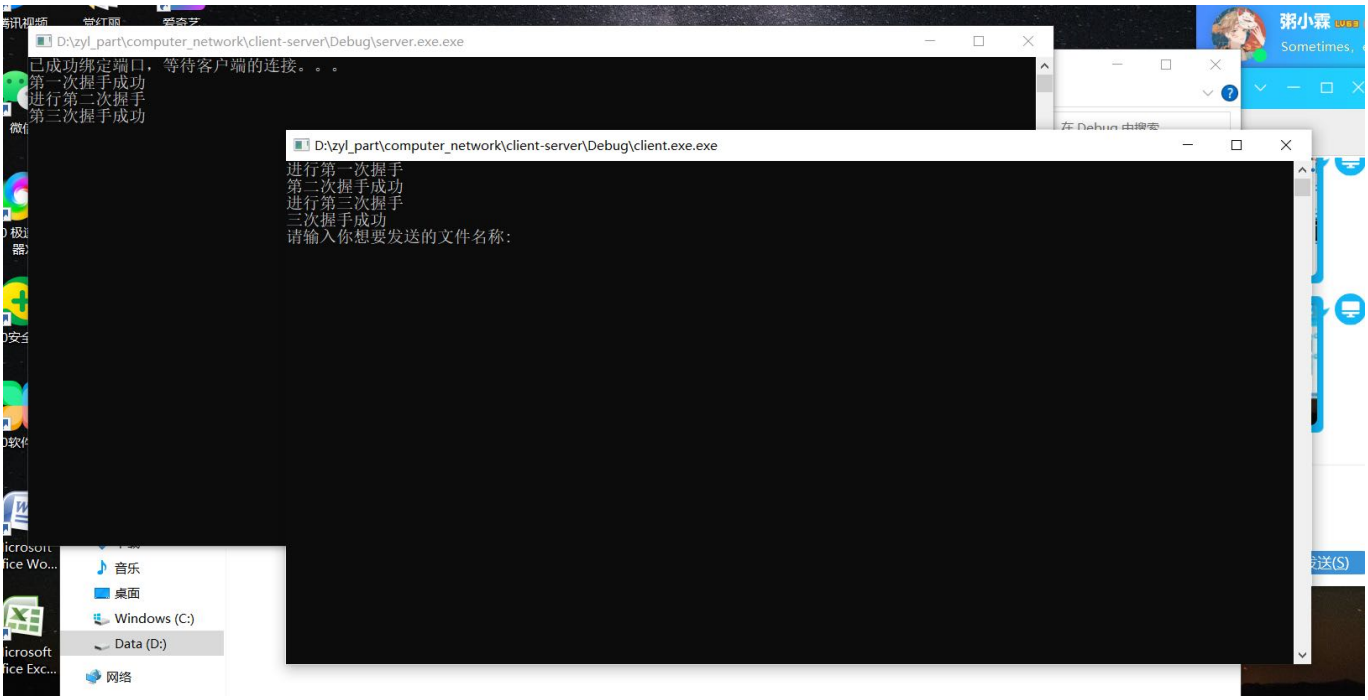
```
void send_file(string file_name, SOCKET* server, SOCKADDR_IN* server_addr)
{
    ifstream file(file_name.c_str(), ifstream::binary);

    // 预定义总长度
    int all_file_len = 0;
    file.seekg(0, file.end);
    all_file_len = file.tellg();
    file.seekg(0, file.beg);

    // 将其放到缓冲区
    int buf_len = file_name.length() + all_file_len + 1;
    char* zyl_buf = new char[buf_len];
    memset(zyl_buf, 0, sizeof(char) * buf_len);
    memcpy(zyl_buf, file_name.c_str(), file_name.length());
```

```
    // 分割文件名和文件内容
    zyl_buf[file_name.length()] = 0;
    file.read(zyl_buf + file_name.length() + 1, all_file_len);
    file.close();
    cout << "正在发送" + file_name << "文件 其大小为: " << all_file_len << "字
节" << endl;
    clock_t start = clock();

    // 先开始接收
    thread receive_t(rec_thread, server, server_addr);

    // 开始发送
    for (int offset = 0; offset < buf_len; offset += MSS)
    {
        send_one(server, server_addr, zyl_buf + offset, buf_len - offset
>= MSS ? MSS : buf_len - offset, buf_len - offset <= MSS ? true : false);
    }
    // 缓冲区还有数据就不退出
    while (GoBackN_buf.size() != 0)
    {
        continue;
    }
    finish_send = true;

    // 回收线程资源
    receive_t.join();
    clock_t end = clock();
    cout << "已成功发完" + file_name + "文件! " << endl;
    // 测试为毛为零
    cout << "start:" << start << ";end:" << end << endl;
    cout << "用时: " << double(end - start) / double(CLOCKS_PER_SEC) << "s"
<< endl;
    cout << "吞吐率: " << double(buf_len) / (double(end - start) /
double(CLOCKS_PER_SEC)) << "byte/s" << endl;
    delete[]zyl_buf;
}
```

四、实验结果

(一)建立连接

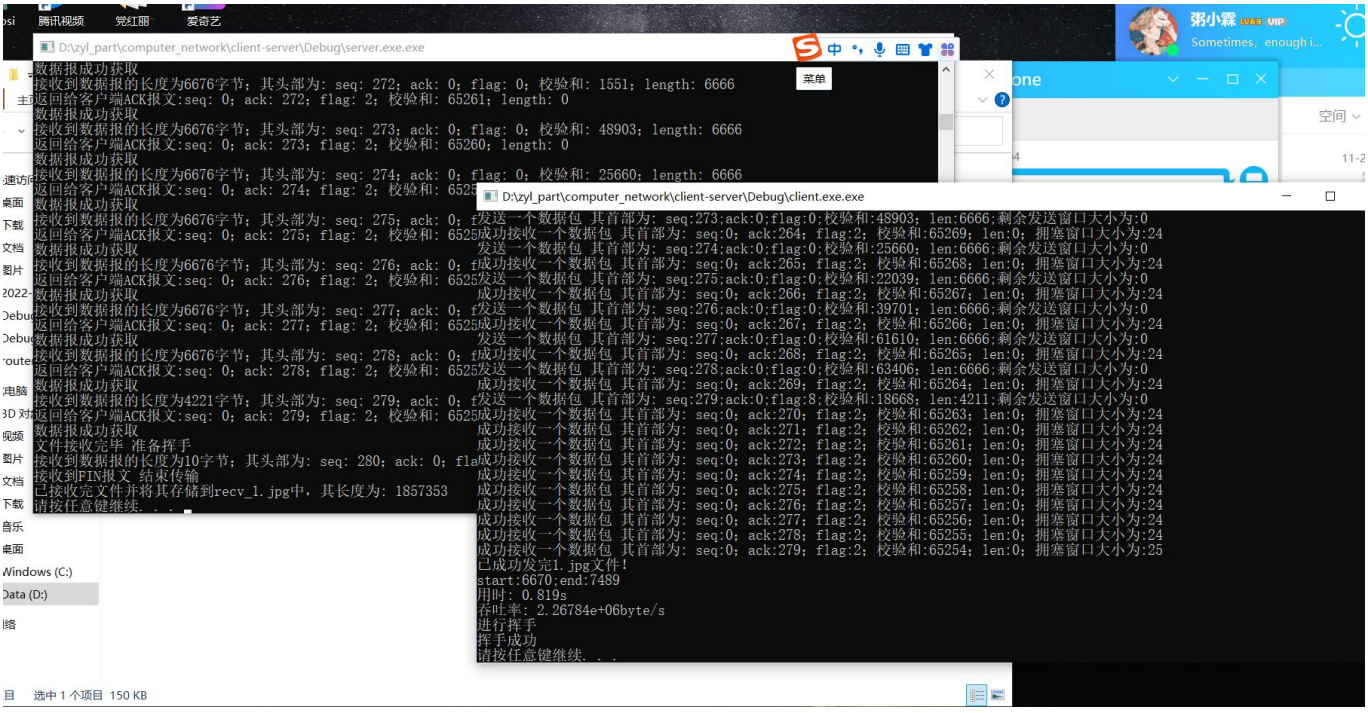
- 首先打开客户端和服务器端，建立三次挥手如下图所示：



(二)数据传输

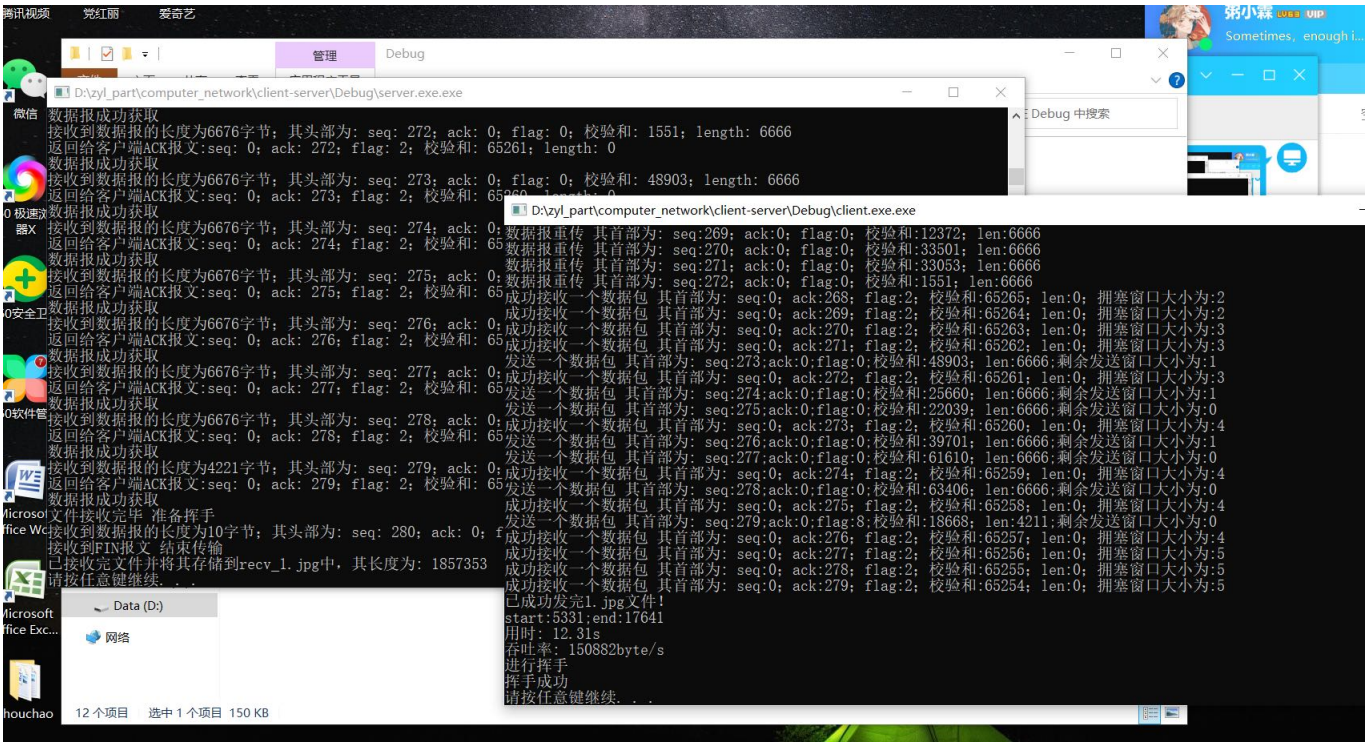
本次传输展示`1.jpg`，过程如下所示：

> 首先是没有丢包和延时的情况：



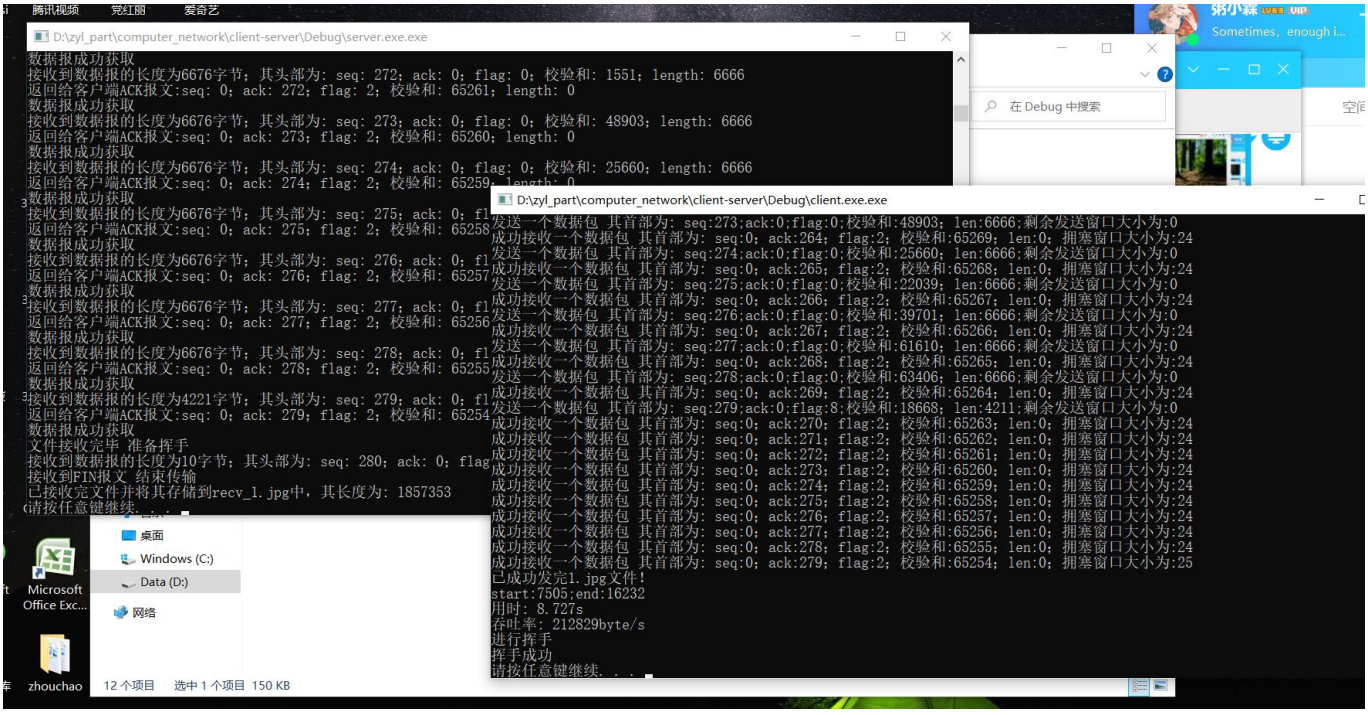
可以看到不到1秒就传完了文件，并且窗口也变换到了25

> 接下来是丢包设置为5%的情况：



可以看到传输时间为12.31秒，明显变慢，并且窗口最后也只有5

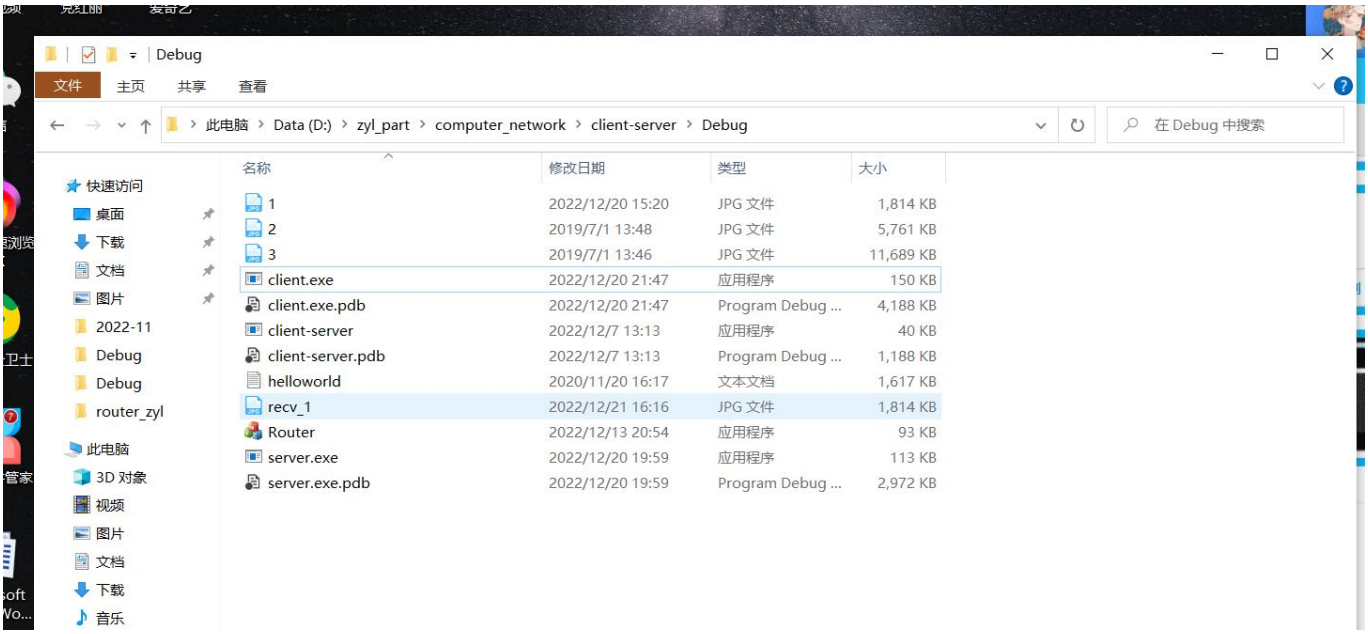
> 最后是延时设置为10ms的情况：



可以看到传输时间为8.727秒，相对于正常网络也变慢许多，并且吞吐量变大

(三)结果展示

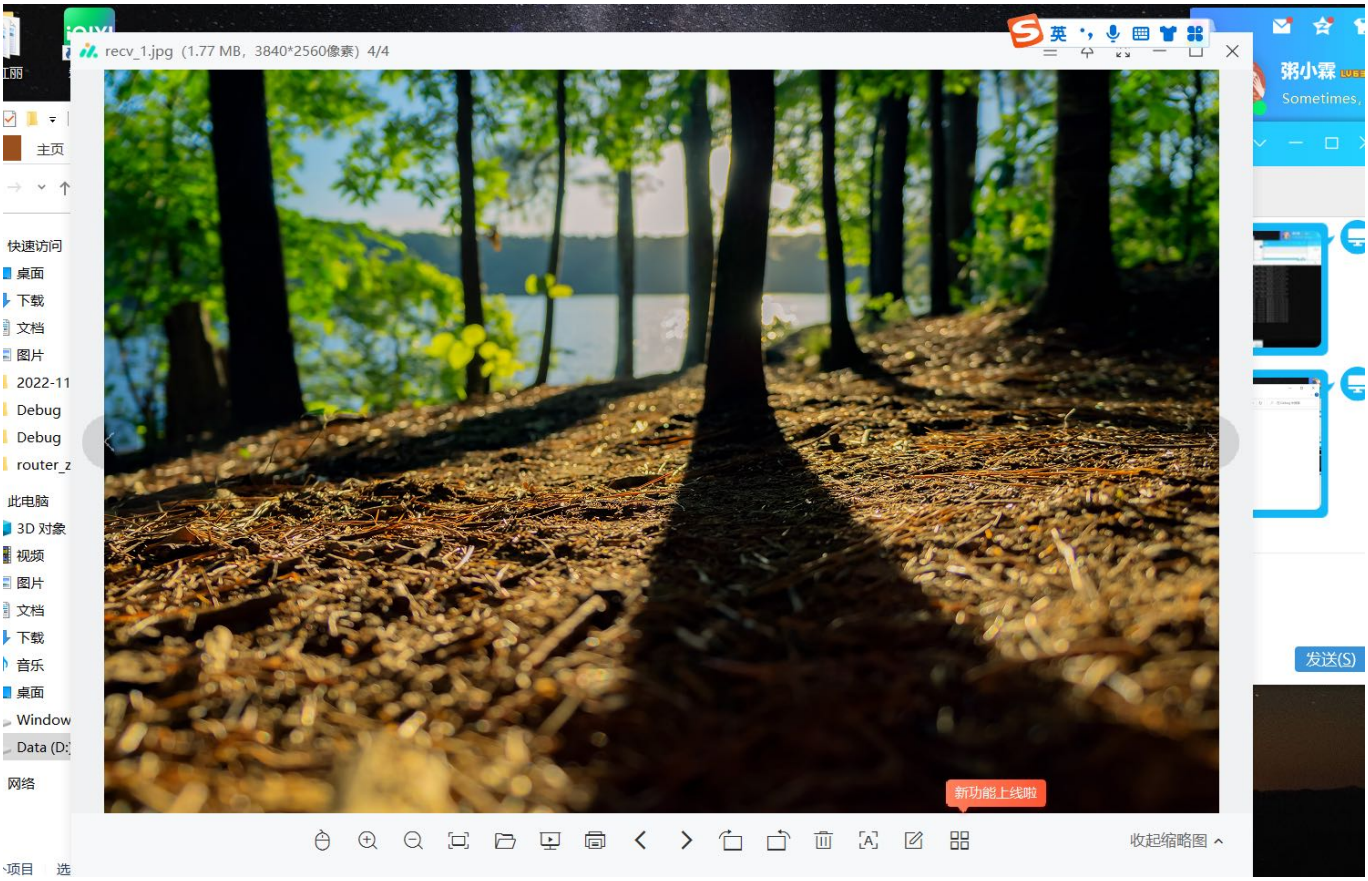
本此展示传输`1.jpg`文件，并且为了区分与原文件的区别，所以在接收的文件存储的时候会加一个`recv_`前缀来表示区分，最后接收到的结果如下图所示：



可以看到有recv_前缀的文件即为成功接收到的文件。

- 接下来查看接收到的内容：

> recv_1.jpg



可以看到接收到的文件都与原文件大小、信息完全相同，传输成功！

可以看出，即使在路由器中设置了丢包或者是延时，但利用拥塞控制Reno算法，传输效率时间也快于实验一，说明用该机制效率远超于实验一

五、问题与思考

(1)问题

本次实验采用多线程进行设计，由于线程的输出时间是不固定的，所以有可能在输出的时候顺序错乱，除此之外缓冲区和计时也有可能错乱

(2)思考

由于以上问题的出现，所以加入了mutex来对程序进行上锁，定义多个类型的mutex变量，用来实现在一处更改（比如输出、缓冲区等）时其他地方不能更改，问题到此也就顺利解决了。

六、总结与展望

(1)总结

本次实验是计算机网络的3-3实验，这一次的实验也实现了多线程的传输，以及实现了reno拥塞控制算法。主要的工作是对多线程的设计，并实现拥塞窗口的变化，通过此次实验让我对网络方面的编程更加的熟悉。

(2)展望

本次实验是实现了拥塞控制，也让本学期的实验接近尾声。本次实验也让我对网络方面的东西产生了更大的兴趣，由于本学期也选上了网络技术与应用这门课，感觉这两门课所用的东西是相辅相成的，希望自己可以结合运用，在本学期得到更好的发展，万事胜意、心想事成、未来可期。