计算机网络第——实验3-3

- 实验名称：基于UDP服务设计可靠传输协议并编程实现（实验3-3）
- 专业：物联网工程
- 姓名：秦泽斌
- 学号：2212005

一、实验要求

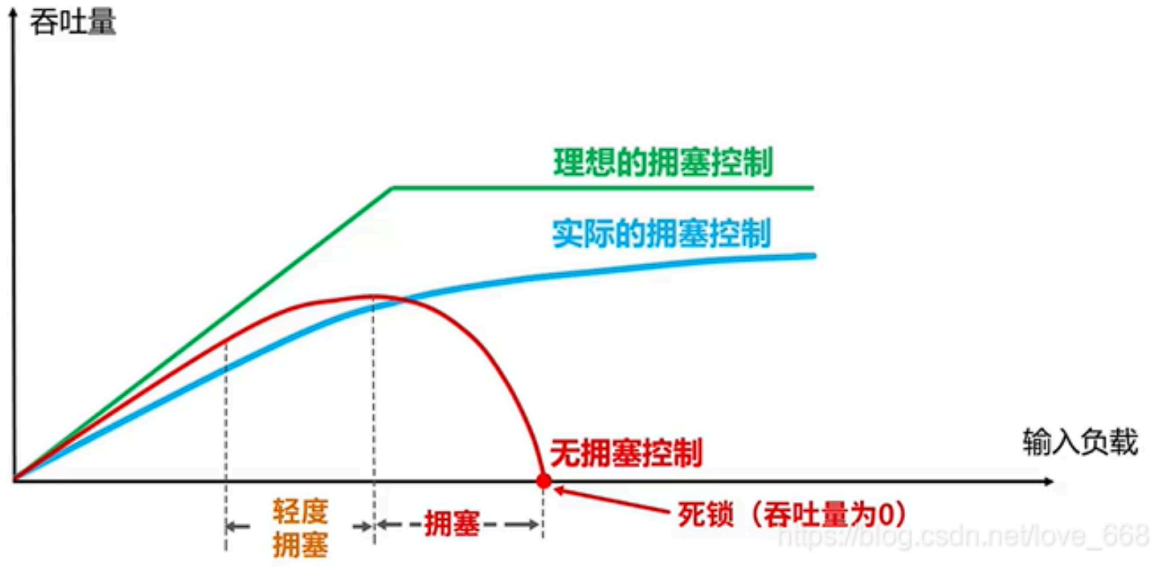
在实验3-2的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

二、实验原理

1.什么是拥塞控制

我们都知道，网络错综复杂，在这个复杂的网络中，很少有两台主机是直接相连。尽管如此，我们还是可以通过网络与其他主机通信，这是为什么？因为我们发送到网络中的数据，当达到网络中的一个节点时（假设是路由器），它会根据数据包含的地址，帮我们将数据转发到离目的地更近的路由器，或直接转发到目的地。但是，这些路由器不是直接就可以转发的，它们需要先将接收到的数据放入自己的内存（可能还要做一些处理），再从中取出进行转发。

这里就面临一个问题：**路由器的内存是有限的，若同一时间到达某个路由器的数据太多，这个路由器将无法接收所有的数据，只能将一部分丢弃；或者同一台路由器数据太多，后面到达的数据将要等待较长的时间才会被转发。**网络中的数据太多，导致某个路由器处理不过来或处理地太慢，这就是**网络拥塞。**若是对于`TCP`这种有重传机制的传输协议，当发生数据丢失时，重传数据将延长数据到达的时间；同时，高频率的重传，也将导致网络的拥塞得不到缓解。**拥塞控制，就是在网络中发生拥塞时，减少向网络中发送数据的速度，防止造成恶性循环；同时在网络空闲时，提高发送数据的速度，最大限度地利用网络资源。**说的简单点，就和堵车差不多，路就这么宽，来的车多了，自然过的就慢，所以在必要的时候要限号。



2. TCP的拥塞控制方法

因为网络层不会提供拥塞的反馈信息，所以**TCP协议采用的是自己判断网络的拥塞情况的方法。**当`TCP`检测到网络拥塞，则降低数据的发送速率，否则增加数据的发送速率。这里就将面临三个问题：

1. `TCP`如何限制数据的发送速率；
2. `TCP`如何检测网络中是否拥塞；
3. `TCP`采用什么算法来调整速率（什么时候调整，调整多少）；

首先来回答第一个问题。了解`TCP`的应该知道，`TCP`不是发送一个数据段，接收到确认后再发送另一个数据段，它采用的是**流水线**的方式。`TCP`的每一个数据段都有一个序号，而`TCP`维护一个**发送窗口**来发送是数据，这个窗口就是一个**区间**。所有序号位于这个窗口内的数据段都会被一次性发送，而不需要等待之前发送的数据段被确认。而每当最早发送出去的数据段被确认，窗口就会向前移动，直到移动到第一个没有被确认的序号，这时候又会有新的数据段序号被包含在窗口中，然后被发送出去。所以限制数据发送速率最好的方式就是**限制窗口的大小。**在发送方的`TCP`程序会跟踪和维护一个叫做**拥塞窗口**的变量，用来进行拥塞控制。拥塞窗口被称为`cwnd`。在`TCP`发送端，所有被发送但是还没收到确认的数据段必须落在这个窗口中，所有，当网络拥塞时，TCP程序将减小`cwnd`，而网络通畅时，增大`cwnd`，以此来控制数据发送的速率。

接着来回答第二个问题。`TCP`程序将通过数据发送的一些现象来推测网络是否拥塞，比如：

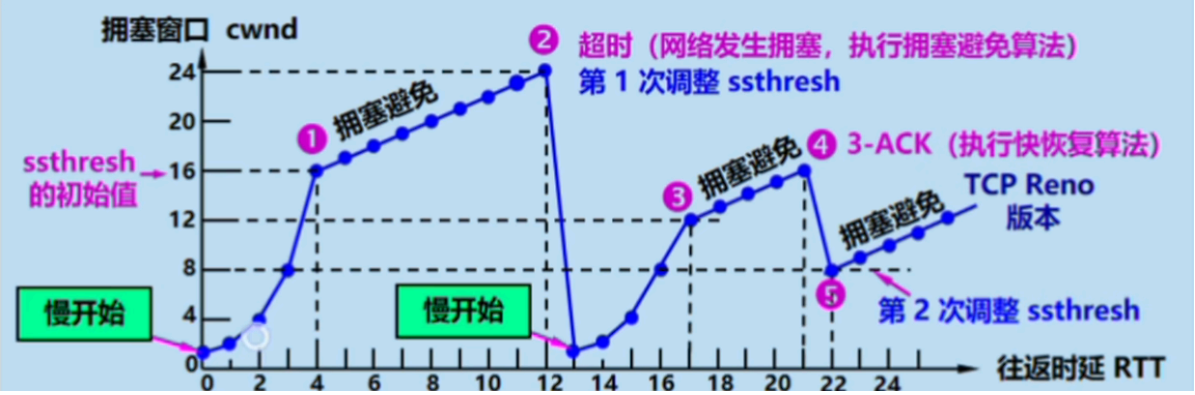
- 若发送一条数据段后，成功接收到了接收方的确认报文，则可以认为网络没有拥塞；
- 若发送出一条数据段后，在规定时间内没有收到确认报文（丢失或时延太大），则可以认为网络出现了拥塞；
- 若连续收到接收方对同一条报文的三次冗余确认（也就是四次确认），则可以推测那条报文丢失，即发生了拥塞；

上面的第三种情况，与`TCP`的**快速重传**机制有关，我这里不详细说明，只是大致介绍一下。TCP无法保证数据能够按顺序到达接收端，所以，可能出现序号靠后的数据报反倒先到达的情况。而`TCP`接收方并不是接收到哪一条报文，就向发送方发送哪条报文的确认，它是通过发送当前应该接收到的序号最小的报文进行确认。举个例子：

1. 发送方同时发送了`1，2，3，4，5`这五个序号的报文，假设接收方接收到序号`1`的报文，于是将向发送方发送一个确认号为`2`的确认报文（`TCP`发送方通过确认号来判断接收方接收到的报文），告诉发送方我已经收到`2`之前的报文了，下一条报文我想要`2`；
2. 接收方接收到`2`号报文后，发送确认号`3`，告诉发送方我接收到了`3`之前的所有报文；
3. 这时候因为网络的不确定性，在`3`号没有到达前，`4`号报文先到达了，接收方接收到后，将它放入缓存，并依旧回复确认号`3`，表示它需要的是3号报文；
4. 可是，之后到达的却`5`，于是它将`5`放入缓存，并依旧回复3；
5. 直到`3`迟迟的到来，这时候接收方接收`3`，同时发现缓存中存在`4和5`，于是回复发送方确认号`6`，表示自己已经接收到了`6`之前的全部报文，下一条需要`6`；

在这里，发送方一共接收到了三条确认号为`3`的确认报文（确认号为`3`是对`2`号报文的确认），第一条正常，后面两条冗余。而所谓的快速重传就是指：**若发送方接收到对同一条报文的三次冗余确认（也就是四次确认），就认为这条报文的下一条已经丢失，于是不管计时器是否超时，都直接重传这条报文的下一条。**上面的例子中，`2`号报文被冗余确认了两次，还不构成快速重传的条件。而为什么是三次，其实就是概率和时间长短的折中选择。

下面回到正题，当快速重传的条件发生，发送方将认为出现了拥塞导致丢包。所有归根到底，**TCP判断拥塞的方式就是检测有没有丢包。**于是我们就可以回答第三个问题了，`TCP`如何调整发送速率——**在没有丢包时慢慢提高拥塞窗口cwnd的大小，当发生丢包事件时，减少cwnd的大小。**当然，具体的算法要复杂的多，`TCP`调整拥塞窗口的主要算法有**慢启动**，**拥塞避免**以及**快速恢复**，其中前两个是`TCP`规范要求必须实现的，而第三个则是推荐实现的，`TCP`根据情况在这三者之间切换。下面我就来一一介绍。



3. 慢启动

在讲解之前，首先得知道一些名词：

- **MSS**：最大报文段长度，`TCP`双方发送的报文段中，包含的数据部分的最大字节数；
- **cwnd**：拥塞窗口，`TCP`发送但还没有得到确认的报文的序号都在这个区间；
- **RTT**：往返时间，发送方发送一个报文，到接收这个报文的确认报文所经历的时间；
- **ssthresh**：慢启动阈值，慢启动阶段，若`cwnd`的大小达到这个值，将转换到拥塞避免模式；

慢启动是建立`TCP`连接后，采用的第一个调整发送速率的算法（或叫模式）。在这个阶段，`cwnd`通常被初始化为`1MSS`，这个值比较小，在这个时候，网络一般还有足够的富余，**而慢启动的目的就是尽快找到上限。**在慢启动阶段，发送方每接收到一个确认报文，就会将`cwnd`增加`1MSS`的大小，于是：

- 初始`cwnd=1MSS`，所以可以发送一个`TCP`最大报文段，成功确认后，`cwnd = 2MSS`；
- 此时可以发送两个`TCP`最大报文段，成功接收后，`cwnd = 4 MSS`；
- 此时可以发送四个`TCP`最大报文段，成功接收后，`cwnd = 8 MSS`.....

由于TCP是一次性将窗口内的所有报文发出，所以所有报文都到达并被确认的时间，近似的等于一个RTT（记住这个结论，后面所述的RTT都是基于这个结论）。所以在这个阶段，拥塞窗口`cwnd`的长度将在每个`RTT`后翻倍，也就是发送速率将以指数级别增长（所以不要被慢启动这个名字误导了）。那这个过程什么时候改变呢，这又分几种情况：

- 第一种：若在慢启动的过程中，发生了数据传输超时，则此时`TCP`将`ssthresh`的值设置为`cwnd / 2`，然后将`cwnd`重新设置为`1MSS`，重新开始慢启动过程，这个过程可以理解为试探上限；
- 第二种：第一步试探出来的上限`ssthresh`将用在此处。若`cwnd`的值增加到`>= ssthresh`时，此时若继续使用慢启动的翻倍增长方式可能有些鲁莽，所以这个时候结束慢启动，改为拥塞避免模式；
- 第三种：若发送方接收到了某个报文的三次冗余确认（即触发了快速重传的条件），则进入到快速恢复阶段；同时，`ssthresh = cwnd / 2`，毕竟发生快速重传也可以认为是发生拥塞导致的丢包，然后`cwnd = ssthresh + 3MSS`；

以上就是慢启动的过程，下面来介绍拥塞避免。

4. 拥塞避免

刚进入这个模式时，`cwnd`的大小近似的等于上次拥塞时的值的一半（这是由进入这个模式的条件决定的），也就是说当前的`cwnd`很接近产生拥塞的值。所以，**拥塞避免是一个速率缓慢且线性增长的过程**，在这个模式下，每经历一个`RTT`（请注意`2.4`中有关`RTT`的结论），`cwnd`的大小增加`1MSS`，也就是说，假设`cwnd`包含`10`个报文的大小，则每接收到一个确认报文，`cwnd`增加`1/10 MSS`。这个线性增长的过程什么时候结束，分为两种情况：

- 第一种：在这个过程中，发生了超时，则表示网络拥塞，这时候，`ssthresh`被修改为`cwnd / 2`，然后`cwnd`被置为`1MSS`，并进入慢启动阶段；
- 第二种：若发送方接收到了某个报文的三次冗余确认（即触发了快速重传的条件），此时也认为发生了拥塞，则，`ssthresh`被修改为`cwnd / 2`，然后`cwnd`被置为`ssthresh + 3MSS`，并进入快速恢复模式；

我们可以看到，慢启动和拥塞避免在接收到三个冗余的确认报文时，处理方式是一样的：判断发生了拥塞，并减小`ssthresh`的大小，但是`cwnd`的大小却不见得有减小多少，这一点让人疑惑。我个人认为是这样，虽然发送方通过接收三次冗余确认报文，判断可能存在拥塞，但是既然可以收到冗余的确认报文，表示拥塞不会太严重，甚至已经不再拥塞，所以对`cwnd`的减小不是这么剧烈。

5. 快速恢复

快速恢复和上面两种模式不太一样，这种模式在TCP规范中并没有强制要求实现，只是一种推荐实现的模式。在快速恢复阶段，每接收到一个冗余的确认报文，`cwnd`就增加`1MSS`，其余不变，而当发生以下两种情况时，将退出快速恢复模式：

- 第一种：在快速恢复过程中，计时器超时，这时候，`ssthresh`被修改为`cwnd / 2`，然后`cwnd`被置为`1MSS`，并进入慢启动阶段；
- 第二种：若发送方接收到一条新的确认报文（不是冗余确认），则`cwnd`被置为`ssthresh`，然后进入到拥塞避免模式；

这里有一个疑问，进入到此模式的条件就是接收到三次冗余的确认报文，判断报文丢失，那为什么再次接收到冗余确认报文时，`cwnd`还是要增长呢？我搜遍网上博客，只在一篇博客中找到一种说法，我认为还是有一定道理的：此时再次收到一条冗余的确认报文，表示发送端发出的报文又有一条离开网络到达了接收端（虽然不是接收端当前想要的一条），这说明网络中腾出了一条报文的空间，所以允许发送端再向网络中发送一条报文。但是由于当前序号最小的报文丢失，导致拥塞窗口`cwnd`无法向前移动，于是只好将`cwnd`增加`1MSS`，于是发送端又可以发送一条数据段，提高了网络的利用率。

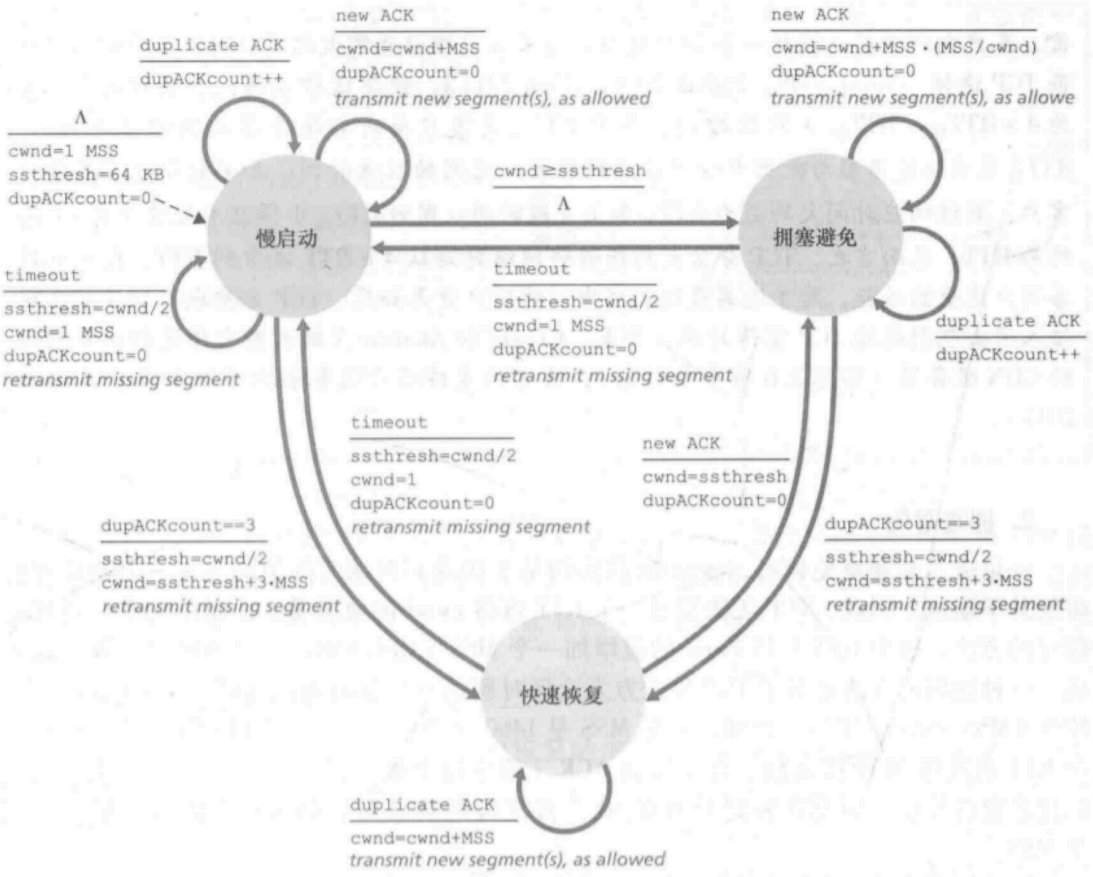


图 3-51　TCP 拥塞控制的 FSM 描述

三、实验内容

1. 慢启动阶段

慢启动阶段的触发有两处，一个是文件刚开始传输时，另一个是发生超时重传后重新慢启动。

文件**开始传输**时，拥塞窗口 `cwnd` 被初始化为1，慢启动阈值 `ssthresh` 被初始化为64.

```
void Reno_init() {
    base = 1;
    next_seqnum = 1;
    ACK_index = 0;
    cwnd = 1;
    ssthresh =64;
    duplicate_ACK_count = 0;
}
```

发生**超时重传**后，拥塞窗口 `cwnd` 被重新设置为1，慢启动阈值 `ssthresh` 被设置为原窗口的一半.

```
// 检查超时并重发
if (clock() - start > MAX_TIME) {
     setConsoleColor(12);
     cout << "[TIMEOUT] 重发未确认数据包." << endl;
     start = clock();
     //超时重传，重新回到慢启动状态,阈值减半
     RTT_ACK = 0;
     ssthresh = cwnd / 2;
     cwnd = 1;
```

```
    setConsoleColor(1);
    cout << "重新进入慢启动阶段，cwnd = " << cwnd << "    ssthresh = " << ssthresh << endl;
    queue<my_udp> temp_queue = message_queue;  // 创建副本用于重发
    while (!temp_queue.empty()) {
        send_packet_GBN(temp_queue.front(), SendSocket, RecvAddr);
        setConsoleColor(7);
        cout << "[RESEND] Packet SEQ=" << temp_queue.front().udp_header.SEQ << " resent." << endl;
        temp_queue.pop();
    }
}
```

接下来时**慢启动的窗口变化**，这段代码被放在接收线程`receive_ACK_thread_Reno()`中，根据理论，在慢启动阶段，窗口随着轮次的增加成指数增长，在代码中就表现为每收到一个正确的ACK，拥塞窗口`cwnd`就+1.

```
// 检查是否是有效的 ACK 包
if (Recv_udp.udp_header.Flag == ACK && checksum((uint16_t*)&Recv_udp, UDP_LEN) == 0) {
    lock_guard<mutex> lock(mtx);  // 保护 base 和队列的访问
    if ((base % DEFAULT_SEQNUM) == Recv_udp.udp_header.SEQ) {
        // 弹出队列中已确认的数据包
        if (!message_queue.empty()) {
            message_queue.pop();
        }
        start = clock();//重置计时器
        base++;  // 滑动窗口前进
        ACK_index++;
        setConsoleColor(7);
        cout << "[ACK RECEIVED] base=" << base << ", ACK_index=" << ACK_index << endl;

        //收到正确的ACK，分为慢启动阶段和拥塞避免阶段两种情况
        FastStatu = 0;//恢复快速重传为可用状态
        FastN = 0;
        duplicate_ACK_count = 0;//重置重复ACK计数
        if (cwnd < ssthresh) {
            cwnd++;//慢启动阶段，每收到一个正确ACK，窗口+1
            RTT_ACK = 0;
            setConsoleColor(1);
            cout << "慢启动阶段，每收到一个正确ACK，窗口+1, cwnd = " << cwnd << "   ssthresh = " << ssthresh << endl;
        }
    }
```

2. 拥塞避免阶段

进入拥塞避免阶段的判断条件是**拥塞窗口`cwnd`>慢启动阈值`ssthresh`**,对于拥塞避免状态中的窗口变化，与慢启动时的指数变化的窗口不同，这时的窗口**随轮次线性增长。**所以在这里，每收到一个正确ACK，+1的不再是拥塞窗口`cwnd`，而是我们新设的一个变量**每轮次中收到的ACK包**`RTT_ACK`，当`RTT_ACK`=`cwnd`时，表明本轮次发送的所有数据包都已经被确认，**这时我们的拥塞窗口`cwnd`才会+1.**

```
// 检查是否是有效的 ACK 包
if (Recv_udp.udp_header.Flag == ACK && checksum((uint16_t*)&Recv_udp, UDP_LEN) == 0) {
    lock_guard<mutex> lock(mtx);  // 保护 base 和队列的访问
    if ((base % DEFAULT_SEQNUM) == Recv_udp.udp_header.SEQ) {
        // 弹出队列中已确认的数据包
        if (!message_queue.empty()) {
            message_queue.pop();
        }
        start = clock();//重置计时器
        base++;  // 滑动窗口前进
        ACK_index++;
        setConsoleColor(7);
        cout << "[ACK RECEIVED] base=" << base << ", ACK_index=" << ACK_index << endl;

        //收到正确的ACK，分为慢启动阶段和拥塞避免阶段两种情况
        FastStatu = 0;//恢复快速重传为可用状态
        FastN = 0;
        duplicate_ACK_count = 0;//重置重复ACK计数
        if (cwnd < ssthresh) {
            cwnd++;//慢启动阶段，每收到一个正确ACK，窗口+1
            RTT_ACK = 0;
            setConsoleColor(1);
            cout << "慢启动阶段，每收到一个正确ACK，窗口+1, cwnd = " << cwnd << "   ssthresh = " << ssthresh << endl;
        }
        else {
            RTT_ACK++;
            if (RTT_ACK == cwnd) {
                cwnd++;//拥塞避免阶段，每收到一个窗口的正确ACK，窗口+1
                RTT_ACK = 0;
                setConsoleColor(1);
                cout << "拥塞避免阶段，每收到一个窗口的正确ACK，窗口+1, cwnd = " << cwnd << "    ssthresh = " << ssthresh << endl;
            }
        }

    }
```

3. 快速恢复

快速恢复阶段的触发条件是**连续三次收到重复冗余ACK包**，便直接触发重传，将所有已发送但未被确认的数据包重新发送，而**不必等到超时后再重传**。在这里我根据实际情况又添加了一点东西，在测试中我发现，当窗口增长到比较大时，由于发送窗口过大，一旦发生丢包便会**触发大量的快速重传**，因为当接收端还没接收到重传过来的数据包时，会接收到大量的上一个窗口余下的数据包，导致接收端会发送大量重复ACK，进而导致触发多次快速重传，这样显然会降低数据通路中的有效信息含量，设置造成死锁。所以我添加了防止多次快速重传的代码，当连续触发时直接视为超时，**重新进入慢启动阶段**，收缩窗口。

```
else {//收到了非预期的ACK包
    setConsoleColor(12);
    cout << "[DUPLICATE ACK] SEQ=" << Recv_udp.udp_header.SEQ << endl;
    duplicate_ACK_count++;
```

```
    //连续收到三次重复ACK包，触发快速重传
    if (duplicate_ACK_count == 3&& FastStatu == 0) {
        FastN++;
        duplicate_ACK_count = 0;
        RTT_ACK = 0;
        ssthresh = cwnd / 2;
        cwnd = ssthresh + 3;
        setConsoleColor(1);
        cout << "快速恢复阶段，cwnd = " << cwnd << "    ssthresh = " << ssthresh << endl;
        start = clock();//重置计时器
        queue<my_udp> temp_queue = message_queue;  // 创建副本用于重发
        while (!temp_queue.empty()) {
            send_packet_GBN(temp_queue.front(), SendSocket, RecvAddr);
            setConsoleColor(7);
            cout << "[RESEND] Packet SEQ=" << temp_queue.front().udp_header.SEQ << " resent." << endl;
            temp_queue.pop();
        }
    }
    if (FastN == 2) {
        FastN = 0;
        FastStatu = 1;//避免连续快速重传
        setConsoleColor(12);
        cout << "[TIMEOUT] 重发未确认数据包." << endl;
        start = clock();
        //超时重传，重新回到慢启动状态,阈值减半
        RTT_ACK = 0;
        ssthresh = cwnd / 2;
        cwnd = 1;
        setConsoleColor(1);
        cout << "重新进入慢启动阶段，cwnd = " << cwnd << "    ssthresh = " << ssthresh << endl;
        queue<my_udp> temp_queue = message_queue;  // 创建副本用于重发
        while (!temp_queue.empty()) {
            send_packet_GBN(temp_queue.front(), SendSocket, RecvAddr);
            setConsoleColor(7);
            cout << "[RESEND] Packet SEQ=" << temp_queue.front().udp_header.SEQ << " resent." << endl;
            temp_queue.pop();
        }
    }
}
```

当然另一种**合理的做法**是在快速重传阶段也添加ACK确认机制，当所有重传的数据包全部被确认后再退出这个阶段，但是由于之前的代码设计中就采用了**在接收线程中设计重传的框架**，整体的大改不太方便，如果可以，以后会尝试实现。（~~最近太忙了bushi~~）

四、实验结果

测试程序丢包率为3%，测试文件为1.jpg，经测试文件成功被接收。

首先，下图展示连接成功和慢启动阶段，每收到一个正确ACK，窗口+1.

```
#####第一次握手成功！ #####
#####第二次握手成功！ #####
#####接收端已成功连接！现在可以发送文件！ #####
test success!
-------------你现在可以使用 quit 命令退出连接！ ------------
-----------------------------------------------------------

#####请输入想要发送的文件名 #####
1.jpg

 #####文件大小： 1857353 bytes
 #####文件名校验和： 24808
 #####送数据包的数量： 1814
 #####滑动窗口大小： 1
已发送数据包0!   数据包大小为 5 bytes
校验和:24808     ACK:16
数据包已确认被接受!      当前发送进度:0/1814   ACK:2

[SEND] Packet SEQ=1 Checksum = 42749
[ACK RECEIVED] base=2, ACK_index=1
慢启动阶段，每收到一个正确ACK，窗口+1, cwnd = 2    ssthresh = 64
[SEND] Packet SEQ=2 Checksum = 4437
[SEND] Packet SEQ=3 Checksum = 55587
[ACK RECEIVED] base=3, ACK_index=2
慢启动阶段，每收到一个正确ACK，窗口+1, cwnd = 3    ssthresh = 64
[SEND] Packet SEQ=4 Checksum = 20059
[SEND] Packet SEQ=5 Checksum = 1678
```

接下来进入拥塞避免阶段，每收到一个轮次的正确ACK（或者称为一个窗口大小的正确ACK），窗口+1.

```
C:\Users\ZZB\Desktop\UDP\3
[ACK RECEIVED] base=725, ACK_index=724
[ACK RECEIVED] base=726, ACK_index=725
[ACK RECEIVED] base=727, ACK_index=726
[ACK RECEIVED] base=728, ACK_index=727
[SEND] Packet SEQ=732 Checksum = 61036
[SEND] Packet SEQ=733 Checksum = 28065
[SEND] Packet SEQ=734 Checksum = 37856
[SEND] Packet SEQ=735 Checksum = 27221
[ACK RECEIVED] base=729, ACK_index=728
拥塞避免阶段，每收到一个窗口的正确ACK，窗口+1, cwnd = 9    ssthresh = 3
[ACK RECEIVED] base=730, ACK_index=729
[ACK RECEIVED] base=731, ACK_index=730
[ACK RECEIVED] base=732, ACK_index=731
[ACK RECEIVED] base=733, ACK_index=732
[ACK RECEIVED] base=734, ACK_index=733
[ACK RECEIVED] base=735, ACK_index=734
[ACK RECEIVED] base=736, ACK_index=735
[SEND] Packet SEQ=736 Checksum = 36949
[SEND] Packet SEQ=737 Checksum = 65262
[SEND] Packet SEQ=738 Checksum = 14668
[SEND] Packet SEQ=739 Checksum = 50176
[SEND] Packet SEQ=740 Checksum = 21227
[SEND] Packet SEQ=741 Checksum = 20527
[SEND] Packet SEQ=742 Checksum = 13796
[SEND] Packet SEQ=743 Checksum = 62273
[SEND] Packet SEQ=744 Checksum = 14681
[ACK RECEIVED] base=737, ACK_index=736
[SEND] Packet SEQ=745 Checksum = 60312
[ACK RECEIVED] base=738, ACK_index=737
拥塞避免阶段，每收到一个窗口的正确ACK，窗口+1, cwnd = 10    ssthresh = 3
```

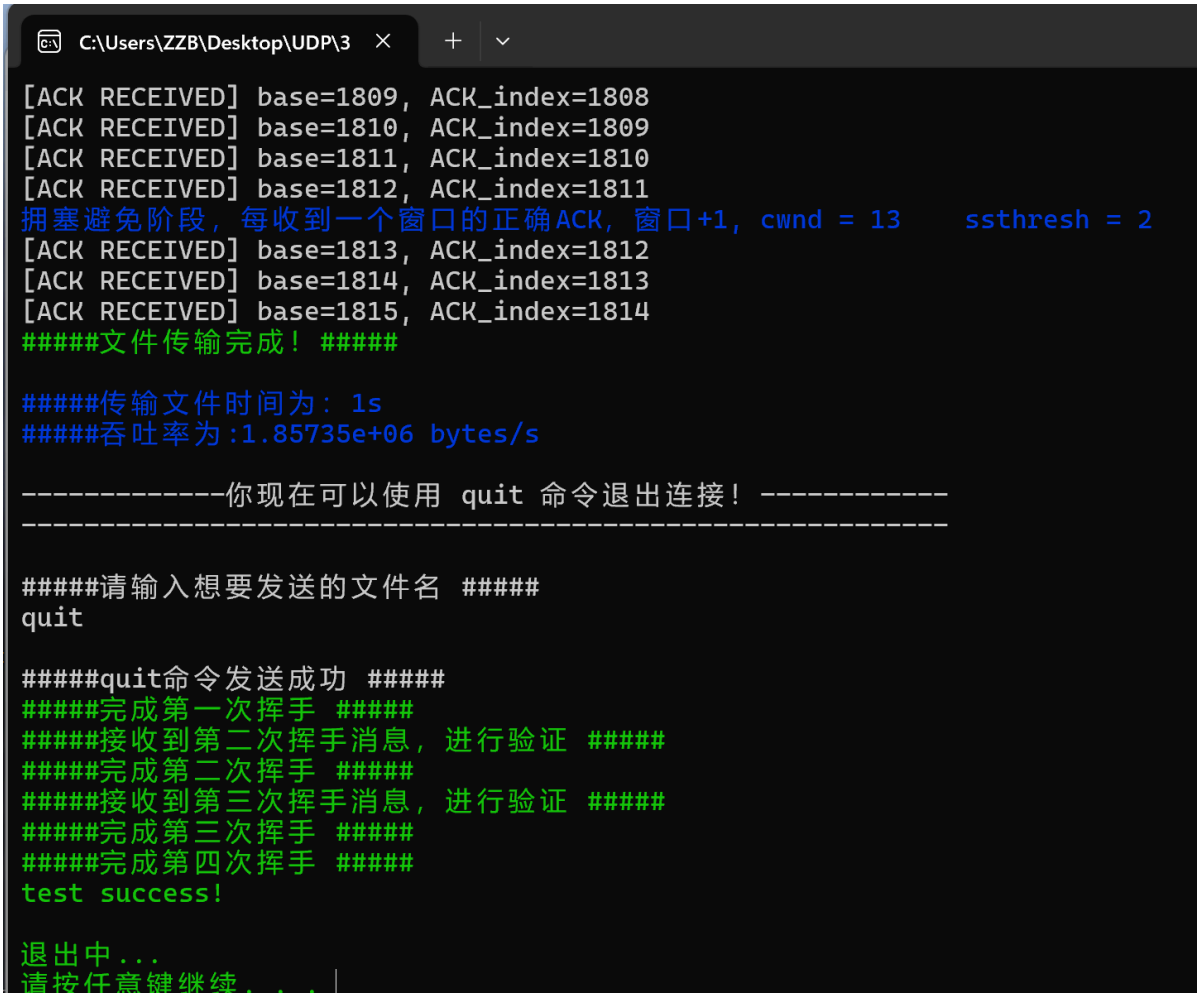
接下来时超时重传后重新进入慢启动阶段的展示，cwnd被设置为1，ssthresh被设置为原窗口的一半。
（这里由于线程之间的打印冲突导致了一些日志的乱序，~~果然应该为打印也单独做一个线程吗~~）

```
[RESEND] Packet SEQ=947 resent.
[TIMEOUT] 重发未确认数据包.
重新进入慢启动阶段, cwnd = 1    ssthresh = 3
[RESEND] Packet SEQ=935 resent.
[RESEND] Packet SEQ=936 resent.
[RESEND] Packet SEQ=937 resent.
[RESEND] Packet SEQ=938 resent.
[RESEND] Packet SEQ=939 resent.
[RESEND] Packet SEQ=940 resent.
[RESEND] Packet SEQ=941 resent.
[RESEND] Packet SEQ=942 resent.
[RESEND] Packet SEQ=943 resent.
[RESEND] Packet SEQ=944 resent.
[RESEND] Packet SEQ=945 resent.
[RESEND] Packet SEQ=946 resent.
[RESEND] Packet SEQ=947 resent.
[DUPLICATE ACK] SEQ=934
[DUPLICATE ACK] SEQ=934
[DUPLICATE ACK] SEQ=934
[DUPLICATE ACK] SEQ=934
[DUPLICATE ACK] SEQ=934
[DUPLICATE ACK] SEQ=934
[ACK RECEIVED] base=936, ACK_index=935
慢启动阶段, 每收到一个正确ACK, 窗口+1, cwnd = 2    ssthresh = 3
[ACK RECEIVED] base=937, ACK_index=936
慢启动阶段, 每收到一个正确ACK, 窗口+1, cwnd = 3    ssthresh = 3
[ACK RECEIVED] base=938, ACK_index=937
[ACK RECEIVED] base=939, ACK_index=938
[ACK RECEIVED] base=940, ACK_index=939
```

最后是快速**恢复阶段**，当接收到连续三个重复冗余ACK后直接触发重传，窗口和阈值都减半。

```
[DUPLICATE ACK] SEQ=707
[DUPLICATE ACK] SEQ=707
[DUPLICATE ACK] SEQ=707
快速恢复阶段, cwnd = 6    ssthresh = 3
[RESEND] Packet SEQ=708 resent.
[RESEND] Packet SEQ=709 resent.
[RESEND] Packet SEQ=710 resent.
[RESEND] Packet SEQ=711 resent.
[RESEND] Packet SEQ=712 resent.
[DUPLICATE ACK] SEQ=707
[ACK RECEIVED] base=709, ACK_index=708
[ACK RECEIVED] base=710, ACK_index=709
[SEND] Packet SEQ=713 Checksum = 239
[SEND] Packet SEQ=714 Checksum = 31597
[SEND] Packet SEQ=715 Checksum = 61552
[ACK RECEIVED] base=711, ACK_index=710
[SEND] Packet SEQ=716 Checksum = 4428
[ACK RECEIVED] base=712, ACK_index=711
[SEND] Packet SEQ=717 Checksum = 2739
[ACK RECEIVED] base=713, ACK_index=712
[SEND] Packet SEQ=718 Checksum = 12315
[ACK RECEIVED] base=714, ACK_index=713
拥塞避免阶段, 每收到一个窗口的正确ACK, 窗口+1, cwnd = 7    ssthresh = 3
[SEND] Packet SEQ=719 Checksum = 32646
[SEND] Packet SEQ=720 Checksum = 16079
[ACK RECEIVED] base=715, ACK_index=714
[ACK RECEIVED] base=716, ACK_index=715
[ACK RECEIVED] base=717, ACK_index=716
[SEND] Packet SEQ=721 Checksum = 16566
[SEND] Packet SEQ=722 Checksum = 62237
```

不要忘记还有传输成功的界面展示。（速度有点快的离谱。。。）



五、总结与分析

在本次实验之中，深入地了解了 **TCP 的拥塞控制算法**包括 Tahoe 算法、Reno 算法以及 New Reno 算法。在实现过程之中，对于多线程以及消息队列的相关知识进行了深入了解。由于 TCP 的机制，我们需要对拥塞控制算法进行一定的改编与优化，在实现完整的设计后可以发现在基于探测思想的拥塞控制算法，在3%丢包率的情形下是3-2效率的三倍多，有了不错的优化效果。

对于在实验中遇到的问题，包括但不限于遇到**死锁，无限超时，无限重传**等各种各样的问题，但在捋清楚思路后逐步调试，问题就都迎刃而解了，收获颇丰也算。