计算机网络实验3-3实验报告

姓名：阿斯雅 学号：2210737

一、协议设计

实验3-2回顾

在3-2实验中，我们实现了基于滑动窗口的可靠性传输机制，即客户端和服务器之间的数据传输通过滑动窗口控制数据的发送和接收，保证数据的可靠性。然而，滑动窗口大小在这一实现中是固定的，并未考虑到网络的拥塞情况。在本次实验3-3中，我们将进一步完善这个机制，引入基于拥塞控制的可靠性传输。

在基于拥塞控制的传输中，滑动窗口的大小不再是固定的，而是根据网络当前的拥塞状况动态调整。客户端将根据超时事件和重复确认（ACK）的出现，来适时地调整窗口大小。具体来说，当客户端检测到超时时，它会认为发生了网络丢包或延迟，进而减小滑动窗口的大小以减轻网络负担；相反，当客户端接收到多个重复的ACK时，它可能会认为网络状况有所改善，从而逐步增大滑动窗口，允许更多数据的并发传输，以提高吞吐量。

通过这种动态调整滑动窗口的方式，客户端能够实时适应网络的拥塞状态，达到更高的传输效率，并减少由于网络过载导致的丢包和延迟问题。这种基于拥塞控制的传输机制能够在动态变化的网络环境中，优化数据传输的性能，确保传输的稳定性和高效性。

拥塞控制

本次实验采用了 TCP Reno 拥塞控制算法，但在深入讲解 Reno 之前，我们首先需要了解其前身——TCP Tahoe 算法，因为 Reno 算法是在 Tahoe 的基础上进行了改进。

Tahoe算法

TCP Tahoe 是一种早期的拥塞控制算法，它通过调整窗口大小来有效管理网络的拥塞，确保数据传输的稳定性。Tahoe 算法通过以下三个主要机制来控制网络拥塞：**慢启动**、**拥塞避免**和**超时**。

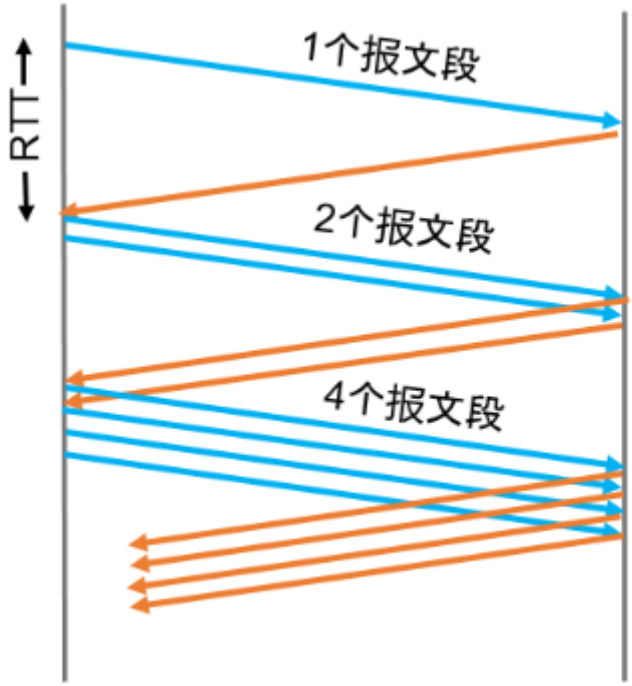
Tahoe 维护两个核心变量：

1. **拥塞窗口 (cwnd)**：表示能够发送的数据量。
2. **慢启动阈值 (ssthresh)**：决定从慢启动到拥塞避免的切换点

1. 慢启动

在 TCP 连接的初期阶段，Tahoe 使用“慢启动”策略。初始时，拥塞窗口 (cwnd) 从一个非常小的初值开始，每收到一个 ACK 确认包后，cwnd 会指数级增长，即每经过一个 RTT（往返时延），cwnd 就会翻倍，直到它的值达到慢启动阈值 (ssthresh)。

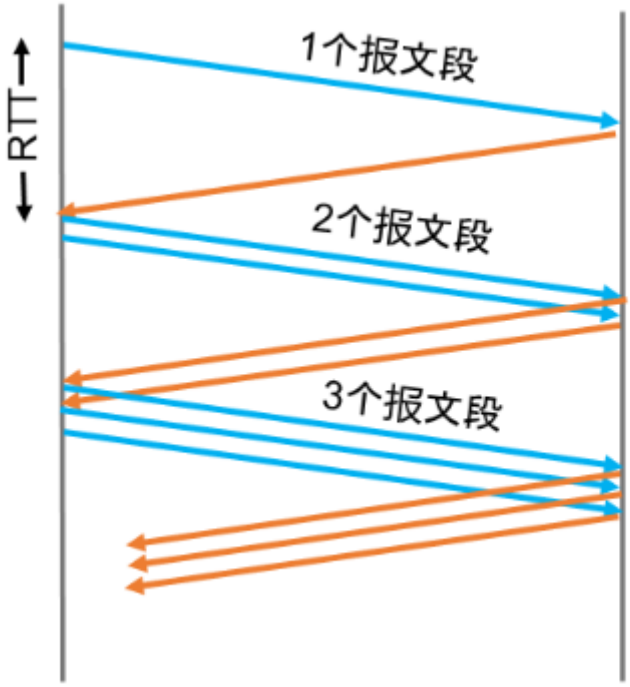
在慢启动阶段，TCP 的发送速率呈指数增长，网络负载会迅速增加，这时候的目标是尽快找出网络的可承载容量。



2. 拥塞避免

当拥塞窗口 (cwnd) 达到慢启动阈值 (ssthresh) 时，TCP 切换到拥塞避免阶段。此时，窗口的增长方式变为线性增长，而不是指数增长。具体来说，每收到一个 ACK 确认包，cwnd 增加 1 个 MSS（最大报文段长度）。

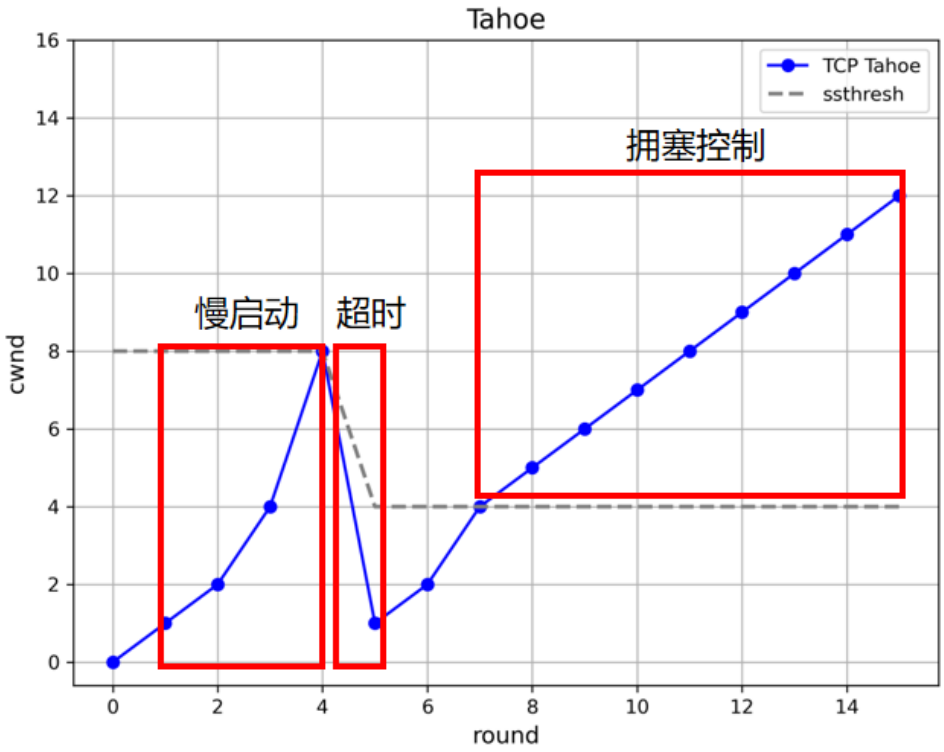
拥塞避免阶段的目的是避免快速增加网络负担，因为此时我们已经接近网络的承载极限。通过减缓窗口的增长，可以降低网络发生拥塞的风险，并提高数据传输的稳定性。



3. 超时

当网络发生拥塞，数据包丢失时，TCP 会触发超时机制。此时，拥塞窗口 (cwnd) 被重置为初始值（通常为 1），并进入慢启动阶段。与此同时，慢启动阈值 (ssthresh) 被设置为丢包时拥塞窗口大小的一半。这个操作能够确保网络不会因为窗口过大而再次发生拥塞，从而提高系统的可靠性。

以下图示反映了随着轮次的变化，拥塞窗口和阈值的演变过程



Reno算法

Reno 算法可以看作是对 Tahoe 的优化。相较于 Tahoe，Reno 引入了一个新机制——**快速恢复**。这个机制使得 Reno 在网络发生丢包时能更高效地恢复传输，而不是像 Tahoe 那样每次丢包后都从慢启动阶段重新开始。

1. 快速恢复

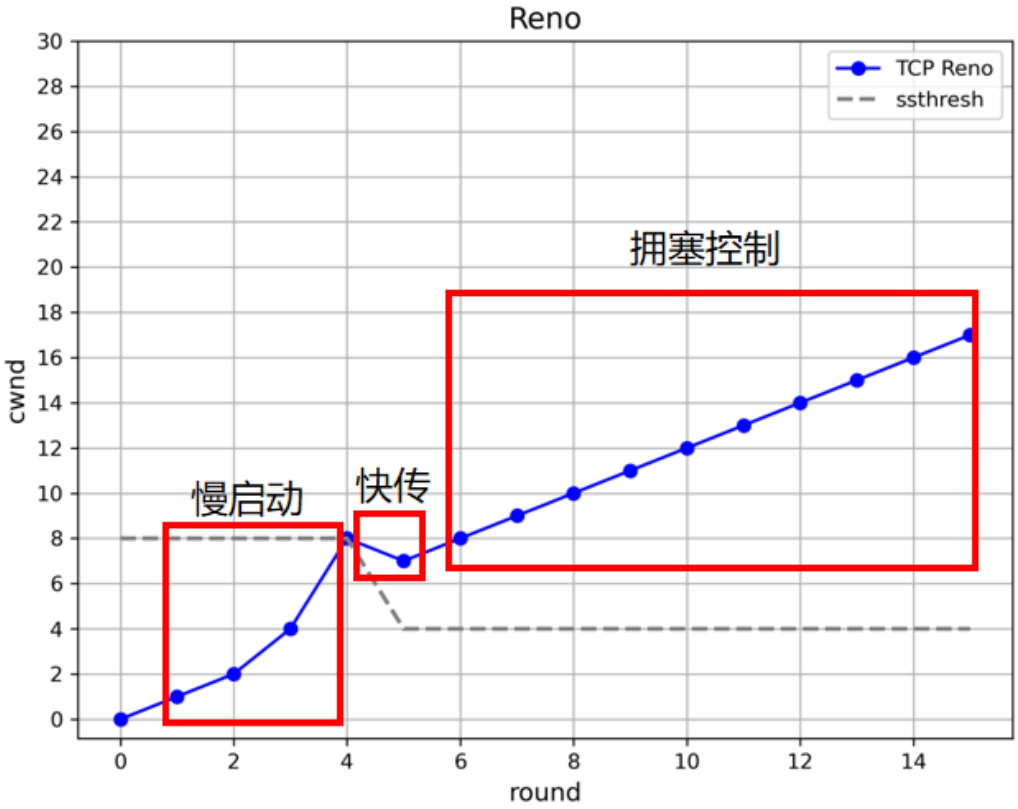
在 Reno 算法中，当接收到三次重复的 ACK（即表示有数据包丢失）时，TCP 不会立即回到慢启动阶段，而是进入**快速恢复**阶段。快速恢复的关键在于，接收到三次重复 ACK 时，表明丢失的数据包的拥塞情况并不严重，因此 TCP 会将 **慢启动阈值 (ssthresh)** 设置为当前拥塞窗口 (cwnd) 的一半，并将 cwnd 调整为 `ssthresh + 3`，这时数据包会立即被重新传输，并且不用像 Tahoe 那样从头开始慢启动。

这种方式让 Reno 更有效地恢复数据传输，同时也避免了过度回退，提高了网络利用率。

2. 其他机制

除了快速恢复外，Reno 还继承了 Tahoe 的慢启动、拥塞避免和超时处理机制，因此在网络丢包较为严重时，Reno 仍会通过超时重传并回到慢启动阶段。而在拥塞避免阶段，cwnd 的增长方式仍然是线性的。

以下是 Reno 算法中随着轮次变化的拥塞窗口和阈值图：



异常情况处理

在 TCP 拥塞控制中，异常情况的处理非常重要，因为它直接影响到数据传输的稳定性和效率。以下将详细讨论在 **客户端丢包** 和 **服务端丢包** 两种异常情况下的处理方式。

客户端丢包

客户端丢包是网络传输中常见的问题，特别是在高延迟或不稳定的网络环境中。当客户端丢失了数据包，通常会导致服务端收到乱序的数据包，从而触发一些恢复机制。TCP Reno 算法中主要通过 **快速重传** 和 **超时重传** 两种方式来解决客户端丢包问题。

1. 快速重传

当客户端发送的某个数据包丢失时，服务端会收到数据包的后续部分，并发送重复的 ACK 给客户端，表示它已经收到了该丢失数据包之后的数据。这些重复 ACK 会帮助客户端检测到丢包并触发 **快速重传** 机制。快速重传的过程如下：

- 当客户端在较短时间内收到多个相同的重复 ACK 时，TCP Reno 认为该数据包丢失了。
- 客户端会立即重新发送丢失的数据包，而不需要等待重传超时。

优势：快速重传的最大优势是它能在网络发生丢包时迅速恢复数据传输，避免了长时间的等待，降低了网络延迟，并减少了网络的拥塞。

2. 超时重传

如果客户端没有收到重复的 ACK，或者由于网络延迟较大导致重复 ACK 数量不足，无法触发快速重传机制，那么客户端就会经历 **超时重传** 机制。这种情况下，客户端会等待一个自适应的超时时间（通常会根据网络延迟进行调整），然后在超时之后重新发送丢失的数据包。

优势与劣势：

- **优势**：超时重传作为一种备用机制，在没有触发快速重传的情况下依然能保证数据的可靠传输。

- **劣势**：由于超时重传需要等待超时时间的到来，因此恢复速度较慢，可能会导致较大的延迟，尤其在网络状况不佳时，重传的恢复效率较低。

服务端丢包

与客户端丢包不同，服务端丢包对数据传输的影响相对较小，因为 TCP 使用了 **累积确认机制**。具体来说，确认包是累积的，即每个确认包都会确认所有小于或等于当前确认序列号的数据包。因此，即使服务端丢失了一个确认包，客户端依然能够通过其他确认包获取必要的确认信息。

1. 累积确认机制的作用

累积确认机制意味着，服务端确认的 ACK 不会仅仅确认单个数据包，而是会确认一个序列号范围内的所有数据包。例如，假设客户端已经发送了多个数据包，并等待服务端的确认：

- 如果服务端丢失了一个确认包，客户端并不会立即感知到这个问题。
- 客户端会继续接收并处理服务端发来的后续确认包，直到获取到有效的确认信息。

即使某个确认包丢失，客户端仍能通过后续的确认包来得到正确的反馈，从而避免了数据的重复传输或超时重传。

2. 服务端确认包丢失的影响

虽然服务端丢失确认包会导致某些 ACK 无法及时到达客户端，但因为 **累积确认** 的存在，丢失的确认包通常不会导致数据传输的严重中断。客户端会继续等待下一个有效的确认包，不会因为单个确认包的丢失而造成数据传输的停滞。

然而，如果确认包丢失非常严重，可能会导致客户端的 **重传超时**，这时客户端会重新发送未收到确认的数据包。由于客户端发送的数据包可能已经正确接收，因此这种重传过程会增加额外的网络负载，但整体上不会影响数据的最终传输。

二、程序设计

本次实验中，主要的修改集中在 **客户端接收线程** 和 **客户端发送线程** 上。其他的函数和功能与前两次实验基本相同，因此在此不再赘述。

发送线程

首先我引入了两个变量，分别对应拥塞窗口和阈值窗口，并且我把他们都设置成了double类型。

```
double cwnd; //拥塞窗口
double sstreshs; //阈值窗口
```

在我的设计中，发送线程负责按顺序发送数据包，而接收线程则负责监控网络的拥塞状况并动态调整拥塞窗口（`cwnd`）的大小。为了保证发送线程与接收线程之间的同步，我的设计确保在接收线程更新拥塞窗口大小之后，发送线程才开始新的数据发送。为此，在发送线程中我加入了动态调整睡眠时长的机制，这样可以根据当前窗口大小和每次发送的数据包数量来动态调整发送速率。

具体来说，发送线程会在每次发送一批数据包后，依据当前的拥塞窗口（`cwnd`）动态计算出睡眠时间。睡眠时间通过 `base_sleep * cwnd` 来调整，确保每次发送数据包的时间间隔能够根据拥塞窗口的大小自动调节，防止过度发送导致的网络拥堵。

在实现中，`base` 表示当前已发送数据包的序列号，而 `cwnd` 是当前的拥塞窗口大小，控制每次发送的数据包数量。发送过程首先锁定发送线程的资源（通过 `sendl.lock()`），然后从窗口中逐个获取数据包并计算其校验和（通过 `setChecksum`），接着记录每个包的定时器起始时间。每当发送一个包时，发送线程通过 `sendto()` 将数据包发送到服务器。如果模拟了丢包（`if (coun == 100)`），则不发送该包，输出丢包信息。每次成功发送数据包后，都会更新计数器，直到发送完当前窗口中的所有数据包。

发送完成后，`sendl.unlock()` 解锁资源，接着根据窗口大小动态计算睡眠时间，控制发送速率。

```
while (base < datanum) {  // 当待发送数据包数量大于当前已发送的包数时，继续发送

    sendl.lock();  // 加锁，保证对窗口中数据包的操作是线程安全的
    int temp = 0;  // 临时变量，用于存储当前循环的终止位置

    // 发送窗口内的数据包
    for (int i = base; i < base + cwnd; i++) {
        if (i >= datanum) {  // 如果超出了数据包的总数，停止发送
            temp = i;  // 记录终止位置
            break;
        }

        struct data& currentPacket = window[i];  // 获取当前窗口中的数据包
        setChecksum(&currentPacket);  // 为当前数据包计算并设置校验和
        packetTimers[i] = std::chrono::steady_clock::now();  // 记录当前包的定时器起始时间

        if (coun == 100) {  // 模拟丢包
            coun = 0;  // 重置计数器
            cout << "丢包 seqnum=" << currentPacket.seqnum << endl;  // 输出丢包的序列号
        }
        else {
            coun++;  // 增加计数器
            cout << "发送 seqnum=" << currentPacket.seqnum << endl;  // 输出发送的数据包序列号
            // 将当前数据包通过 UDP 发送到服务端
            sendto(clientSocket, (char*)&currentPacket, sizeof(sendData), 0,
               (SOCKADDR*)&servAddr, sizeof(SOCKADDR));
        }
    }

    sendl.unlock();  // 释放锁

    if (temp >= datanum) {  // 如果数据包已经全部发送完毕，跳出循环
        break;
    }

    // 动态调整睡眠时长，基于发送窗口大小(cwnd)
    int dynamic_sleep_time = base_sleep * cwnd;
    Sleep(dynamic_sleep_time);  // 睡眠一定的时间，以控制发送速率
}
```

接受线程

接收线程的主要职责是调整拥塞窗口（cwnd）的大小。具体而言，当cwnd小于慢启动阈值（ssthresh）时，表示系统处于慢启动阶段。在这一阶段，每当接收到一个ACK时，cwnd会增加1。这种机制确保了在成功发送一批数据包并收到对应ACK的确认后，拥塞窗口的大小能够呈指数增长，从而有效地利用网络带宽，提高数据传输效率。

```
if (cwnd < sstreshs) {
     cwnd+= 1;
     SetConsoleColor(10);
     cout << "慢启动状态，窗口大小：" << cwnd << endl;
     cout << "慢启动状态，阈值大小：" << sstreshs << endl;
     SetConsoleColor(7);
}
```

当cwnd大于慢启动阈值（ssthresh）时，系统进入拥塞避免阶段。在这一阶段，每接收到一个ACK，cwnd的增量为1/cwnd。这种设计的目的是为了在网络达到稳定状态时，以较小的步幅增加拥塞窗口，避免过快的窗口增长导致网络拥塞。具体来说，当一批数据包全部发送并收到对应ACK时，拥塞窗口的大小会增加1，确保网络负载平稳增长。

```
else if (cwnd >=sstreshs) {
     cwnd += 1.0 / cwnd;
     SetConsoleColor(10);
     cout << "拥塞控制状态，窗口大小：" << cwnd << endl;
     cout << "拥塞控制状态，阈值大小：" << sstreshs << endl;
     SetConsoleColor(7);
 }
```

如果收到重复的ACK，我使用一个字典来记录每个数据包接收到的重复ACK次数。当某个数据包的重复ACK计数达到3次时，表示该数据包可能丢失，此时需要触发快速重传机制，重新发送该数据包。此外，在执行快速重传后，会根据当前的网络状况更新拥塞窗口（cwnd）和慢启动阈值（ssthresh），通过调整这两个参数来避免进一步的拥塞，并确保数据传输的稳定性。

```
dupAckCouns[base]++;
if (dupAckCouns[base] == 3) {
   cout << "重传" << base << endl;
   sendto(clientSocket, (char*)&window[base], sizeof(sendData), 0,
          (SOCKADDR*)&servAddr, sizeof(SOCKADDR));
   packetTimers[base] = std::chrono::steady_clock::now();
   sstreshs = cwnd / 2;
   cwnd = sstreshs + 3;
   SetConsoleColor(10);
   cout << "快速重传，窗口大小：" << cwnd << endl;
   cout << "快速重传，阈值大小：" << sstreshs << endl;
   SetConsoleColor(7);
 }
```

下面是完整的接受线程代码：

```
// 接收 ACK 的线程函数
// 该函数负责接收来自服务端的 ACK 包并根据 ACK 的值进行处理，
// 主要涉及的处理逻辑包括：更新窗口大小、进行快速重传、调整拥塞窗口等。
void receiveack() {

    while (true) {
        // 清空接收数据缓冲区
        memset(&receiveData, 0, sizeof(sendData));

        // 从服务端接收数据包（ACK）
        recvfrom(clientSocket, (char*)&receiveData, sizeof(receiveData), 0,
            (SOCKADDR*)&servAddr, &addrlen);

        // 加锁，保证线程安全
        sendl.lock();

        // 如果接收到的 ACK 为 1 且 ACK 序列号大于当前的 base，表示一个新的有效的 ACK 被收
到
        if (receiveData.ack == 1 && receiveData.acknum > base) {
            // 重置当前 base 的重复 ACK 计数器
            dupAckCouns[base] = 0;

            // 更新 base 为接收到的 ACK 序列号
            base = receiveData.acknum;
            cout << "收到累积 ACK：acknum=" << receiveData.acknum << endl;

            // 重置重复 ACK 计数器
            dupAckCount = 0;

            // 如果 base 大于等于数据量，表示所有数据包已被确认，结束接收
            if (base >= datanum) {
                stoptimer = true;
                return;
            }

            // 删除已经确认的数据包计时器
            packetTimers.erase(base - 1);

            // 判断当前是否处于慢启动阶段
            if (cwnd < sstreshs) {
                // 如果处于慢启动阶段，窗口大小指数增长
                cwnd += 1;
                SetConsoleColor(10);  // 设置控制台颜色
                cout << "慢启动状态，窗口大小：" << cwnd << endl;
                cout << "慢启动状态，阈值大小：" << sstreshs << endl;
                SetConsoleColor(7);  // 恢复控制台颜色

            }
            // 如果处于拥塞避免阶段，窗口大小线性增长
            else if (cwnd >= sstreshs) {
                cwnd += 1.0 / cwnd;
                SetConsoleColor(10);  // 设置控制台颜色
                cout << "拥塞控制状态，窗口大小：" << cwnd << endl;
                cout << "拥塞控制状态，阈值大小：" << sstreshs << endl;
                SetConsoleColor(7);  // 恢复控制台颜色
            }

            // 删除已确认的包的计时器
```

```
            packetTimers.erase(base - 1);
        }
        else {
            // 如果接收到重复的 ACK，增加重复 ACK 计数器
            dupAckCouns[base]++;

            // 如果重复 ACK 数量达到 3，触发快速重传
            if (dupAckCouns[base] == 3) {
                cout << "重传数据包：" << base << endl;

                // 发送丢失的数据包进行重传
                sendto(clientSocket, (char*)&window[base], sizeof(sendData), 0,
                    (SOCKADDR*)&servAddr, sizeof(SOCKADDR));

                // 记录重传数据包的时间
                packetTimers[base] = std::chrono::steady_clock::now();

                // 调整慢启动阈值和窗口大小
                sstreshs = cwnd / 2;
                cwnd = sstreshs + 3;

                SetConsoleColor(10);  // 设置控制台颜色
                cout << "快速重传，窗口大小：" << cwnd << endl;
                cout << "快速重传，阈值大小：" << sstreshs << endl;
                SetConsoleColor(7);  // 恢复控制台颜色
            }
        }

        // 解锁，释放对资源的访问
        sendl.unlock();
    }
}
```

超时线程

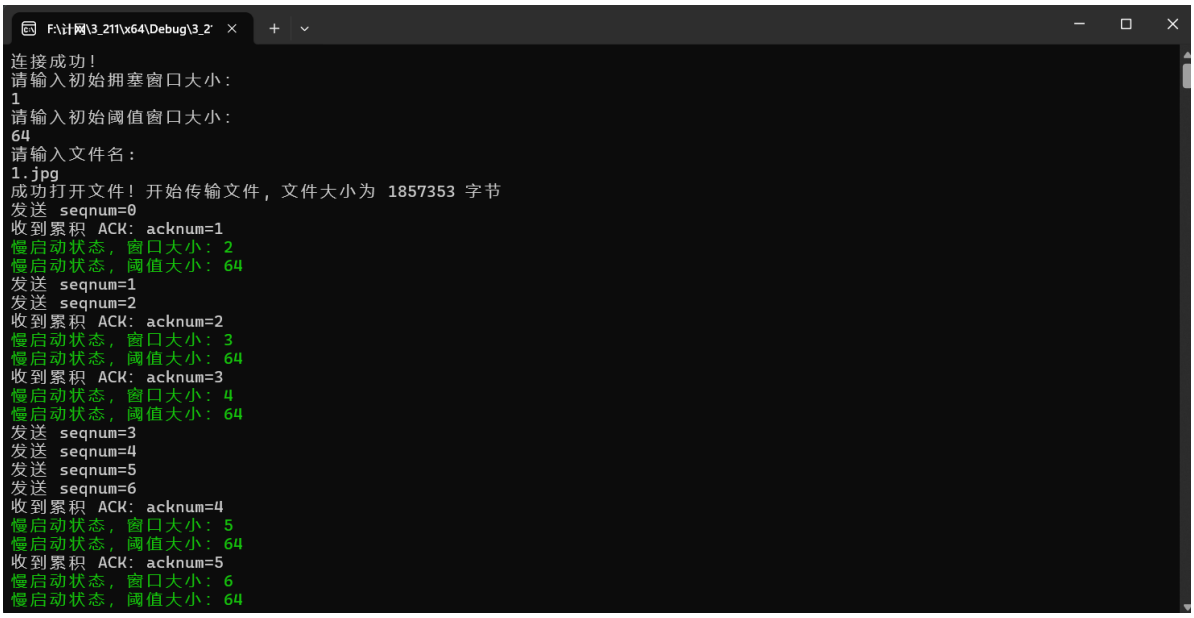
在这个线程里面会一直检查有没有数据包超时，如果有超时数据包，就要进行重传，并且更新拥塞窗口和阈值窗口回到慢启动状态。

```
auto now = std::chrono::steady_clock::now();
if (std::chrono::duration_cast<std::chrono::milliseconds>(now -
packetTimers[i]).count() > TIMEOUT) {
     cout << "超时重传 seqnum=" << window[i].seqnum << endl;
     sstreshs = cwnd / 2;
     cwnd = 1;
     cout << "慢启动状态，窗口大小：" << cwnd << endl;
     cout << "慢启动状态，阈值大小：" << sstreshs << endl;
     sendto(clientSocket, (char*)&window[i], sizeof(sendData), 0,
          (SOCKADDR*)&servAddr, sizeof(SOCKADDR));
     packetTimers[i] = std::chrono::steady_clock::now();
}
```

三、实验结果

下面我以1.jpg为例，选择刚开始的拥塞窗口大小为1，阈值窗口大小为64，运行我的程序：

可以看到刚开始cwnd为1，然后每接到一个ACK包，拥塞窗口大小会加1，而阈值窗口不变化，等到接受线程处理完ACK包之后，发送线程才接着根据更新后的cwnd来发送数据包。

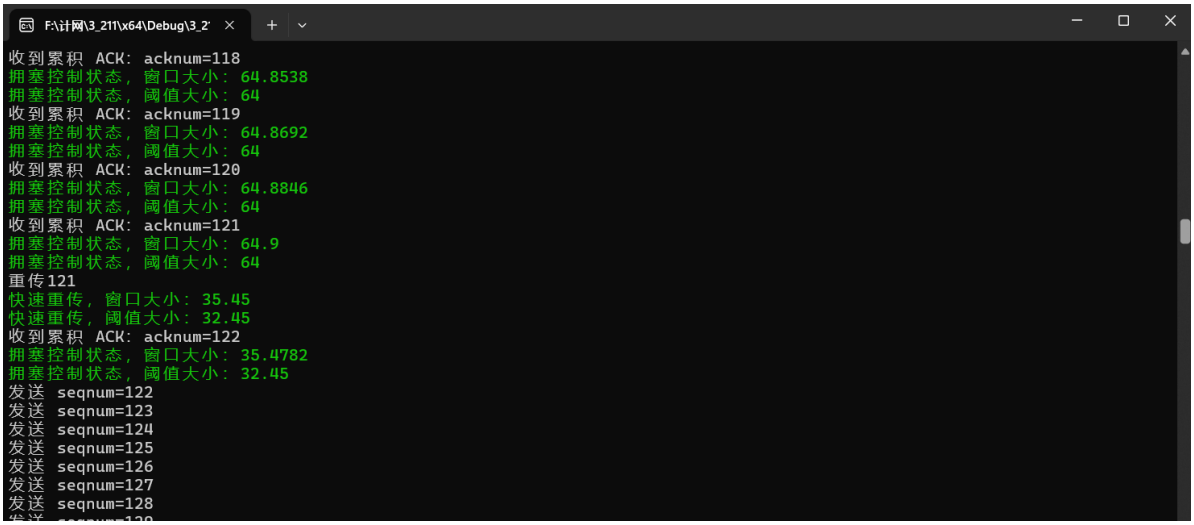


而当cwnd大于等于sstreshs大小的时候,这时候就进入了拥塞避免阶段，cwnd的更新就不会那么快，每接到一个ACK包，会加自己的倒数。

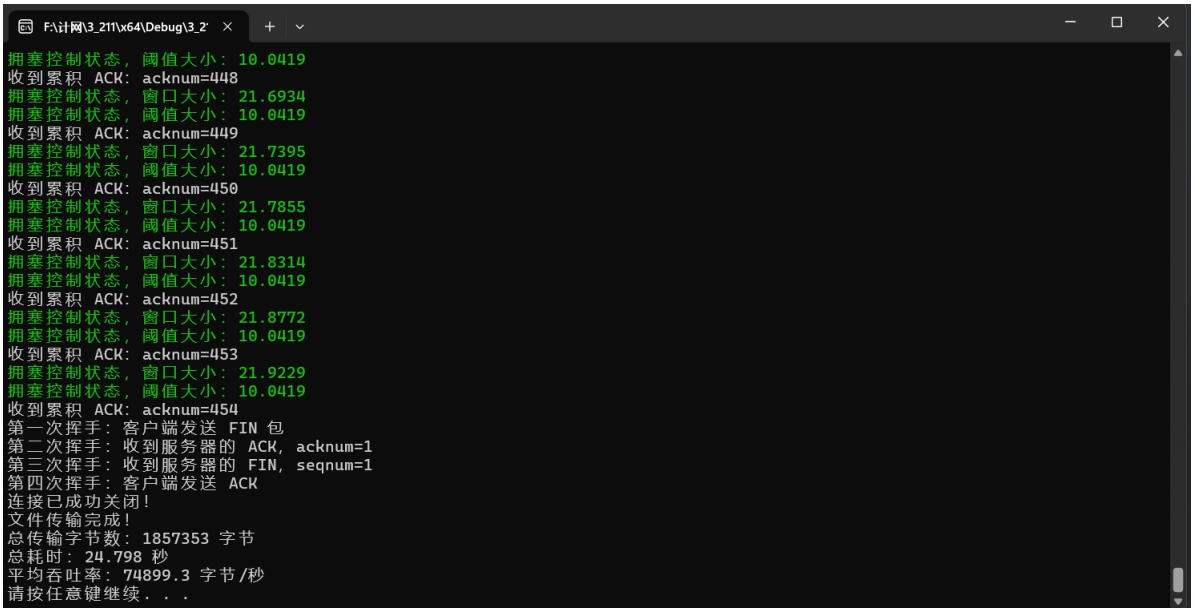
当出现快速重传时，比如图中的121号包，这时候就进入了快传阶段，这时候需要更新cwnd和sstreshs，具体来说：

sstreshs=cwnd/2

cwnd=sstreshs+3



最后断开连接输出统计结果。



四、实验总结

通过本次实验，我深入学习并认识到了TCP协议中的两种主要拥塞控制算法：TCP Tahoe和TCP Reno。通过理论学习和实践操作，我了解了这两种算法如何通过慢启动、拥塞避免、快速重传和快速恢复机制来控制网络中的拥塞。尤其是，Reno算法在Tahoe算法的基础上对快速恢复机制进行了优化，使得网络性能得到了进一步提升。

在实验过程中，Taho算法和Reno算法的具体实现与行为让我深刻理解了它们的区别和应用场景。例如，在面对丢包时，Tahoe会进行全局重传，而Reno则会通过快速重传机制，仅重传丢失的数据包，从而提高了网络的效率。此外，TCP Reno在快速恢复阶段允许拥塞窗口逐渐增加，减少了网络恢复的时间，相比之下，Tahoe则更为保守，导致其恢复过程相对较慢。

通过编程实现这两种算法，我不仅加深了对TCP拥塞控制机制工作原理的理解，还观察到不同算法在网络环境变化时的实际表现。例如，在模拟不同丢包率和网络延迟的条件下，TCP Reno在处理丢包后的恢复效率明显优于TCP Tahoe，这让我更加意识到优化拥塞控制对提高网络性能的重要性。

此外，实验还让我体会到了模拟实验与理论知识之间的紧密联系。通过实验，我能够直观地看到网络性能随延时，丢包率等参数变化的趋势，这对我理解TCP协议的机制及其优化有了更为深刻的认识。

通过本次实验，我也进一步完善了UDP上实现可靠性传输的编程部分。从最初的连接建立、断开连接、超时重传，到GBN滑动窗口机制，再到今天的拥塞控制机制，每一环节都让我更加明白可靠性传输的复杂性和TCP协议在不同网络条件下的适应性。这一过程不仅提高了我的编程能力，还帮助我更全面地理解了TCP协议的设计理念。