



南开大学
Nankai University

南开大学

计算机学院和网络空间安全学院

《计算机网络》实验报告

Lab03-03: 拥塞控制算法

姓名：陆皓喆

学号：2211044

专业：信息安全

指导教师：徐敬东、张建忠

2024 年 12 月 13 日

目录

1 实验要求	2
2 github 仓库	2
3 实验原理	2
3.1 拥塞控制窗口	2
3.2 慢启动阶段	3
3.3 拥塞避免阶段	3
3.4 丢失检测 & RENO 算法	4
3.4.1 超时处理机制	4
3.4.2 RENO 算法	4
4 实验环境	5
5 UDP 报文设计	5
6 协议设计	6
6.1 拥塞控制 & 快速重传	6
6.2 三次握手	6
6.3 客户端 & 服务端数据传输	7
6.4 两次挥手	7
7 功能实现 & 代码分析	7
7.1 Server.h&Server.cpp	7
7.2 Client.h	7
7.3 Client.cpp	8
8 结果展示	12
8.1 无丢包测试	12
8.2 有丢包测试	13
8.3 三次握手测试	14
8.4 两次挥手测试	15
9 心得体会	15

1 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。

- 协议设计：数据包格式，发送端和接收端交互，详细完整
- 发送缓冲区、接收缓冲区
- **RENO 算法或者自行设计其他拥塞控制算法**
- 日志输出：收到/发送数据包的序号、ACK、校验和等，发送端和接收端的窗口大小等情况，传输时间与吞吐率
- 测试文件：必须使用助教发的测试文件 1.jpg、2.jpg、3.jpg、helloworld.txt。

2 github 仓库

本次实验的有关代码和文件，都已经上传至我的个人 github 中。
您可以通过访问[此链接](#)来查阅我的代码文件。

3 实验原理

3.1 拥塞控制窗口

拥塞控制窗口（Congestion Window, cwnd）是用于控制发送方发送数据速率的一个重要参数。它的主要目的是避免网络拥塞，确保网络的高效稳定运行。

为了保证我们的传输速率，我们采用基于窗口的方法，通过拥塞窗口的增大或减小控制发送速率。

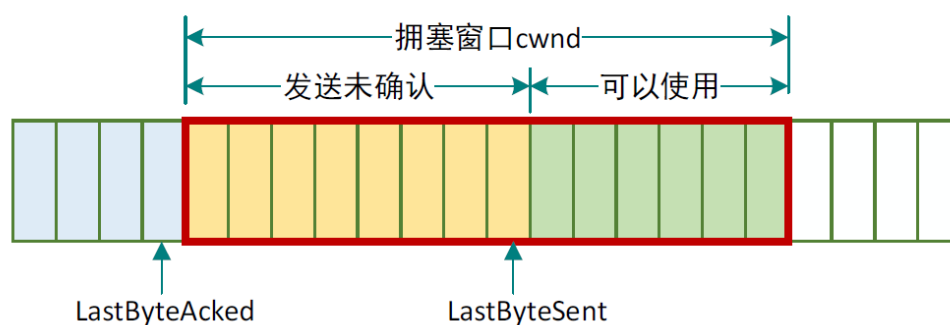


图 3.1: 拥塞窗口 cwnd

对于拥塞窗口的控制，我们需要满足以下的条件：

$$\text{LastByteSent} - \text{LastByteAked} \leq \text{CongestionWindow}(\text{cwnd})$$

所以，总结来说，实际上发送窗口的大小取决于接收通告的窗口和拥塞控制窗口中较小值。

3.2 慢启动阶段

在实验开始时，拥塞窗口大小通常初始化为一个较小的值，一般是 1 个或几个最大报文段长度 (MSS, Maximum Segment Size)。例如，假设初始 $cwnd = 1$ MSS。

发送方每收到一个对新发送数据的确认 (ACK)，就会将拥塞窗口大小增加 1 个 MSS。这就如同一个加速的过程，因为收到的 ACK 数量会随着已发送但未确认的数据段增加而增加。例如，发送方发送了 1 个 MSS 的数据，收到 1 个 ACK 后， $cwnd$ 变为 2 MSS；再发送 2 个 MSS 的数据，收到这 2 个 ACK 后， $cwnd$ 变为 4 MSS，依此类推。在此阶段， $cwnd$ 的增长呈指数形式，其数学表达式为 $cwnd = 2^n$ ，其中 n 是往返时间 (RTT, Round - Trip Time) 的次数，具体的传输图如下所示：

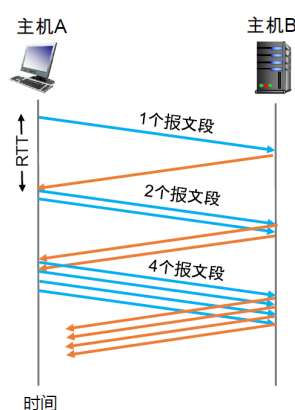


图 3.2: 慢启动阶段

3.3 拥塞避免阶段

当拥塞窗口 ($cwnd$) 大小达到一个称为慢启动门限 ($ssthresh$) 的值时，**TCP 就会从慢启动阶段进入拥塞避免阶段。**

在拥塞避免阶段， $cwnd$ 不再像慢启动阶段那样呈指数增长，而是线性增长。具体来说，每经过一个往返时间 (RTT)， $cwnd$ 增加 1 个最大报文段长度 (MSS)。图中显示了随着时间的推移，每个 RTT 报文段数量逐步增加，即 $cwnd$ 在逐步增加。从图中可以看到，在主机 A 和主机 B 的通信过程中，随着 RTT 的增加，报文段从 1 个增加到 2 个，再增加到 3 个，这体现了 $cwnd$ 的线性增长。

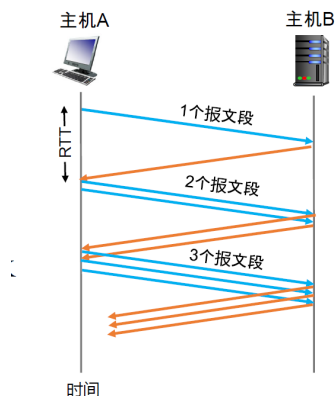


图 3.3: 拥塞避免阶段

TCP 使用字节计数，当收到 ACK 时，拥塞窗口计算如下：

$$cwnd = cwnd + MSS \cdot \frac{MSS}{cwnd}$$

这个公式用于在拥塞避免阶段精确地计算 cwnd 的增长。其中，cwnd 代表拥塞窗口的大小，MSS 表示最大报文段长度。

3.4 丢失检测 & RENO 算法

3.4.1 超时处理机制

当发送方在一定时间内没有收到对已发送数据的确认 (ACK) 时，就认为发生了数据丢失，这种情况称为超时。我们有对应的处理机制：

- 阈值 *ssthresh* (慢启动门限) 被设置为当前拥塞窗口 *cwnd* 的一半，即 $ssthresh = cwnd/2$ 。
- 同时，拥塞窗口 *cwnd* 被重置为 1 个最大报文段长度 (*MSS*)，即 $cwnd = 1(MSS)$ ，然后进入慢启动阶段。

3.4.2 RENO 算法

当接收方收到失序的报文段时，会发送重复的 ACK。如果发送方连续收到三个重复的 ACK，就认为有报文段丢失。我们有对应的处理机制：

- 阈值 *ssthresh* 被设置为当前拥塞窗口 *cwnd* 的一半，即 $ssthresh = cwnd/2$ 。
- 拥塞窗口 *cwnd* 被设置为 $ssthresh + 3(MSS)$ ，然后进入线性增长 (拥塞避免) 阶段。
- 收到重复的 ACK，意味着网络仍然可以交付一些报文段，说明拥塞情况不严重。

具体的 RENO 算法的状态机如下所示：

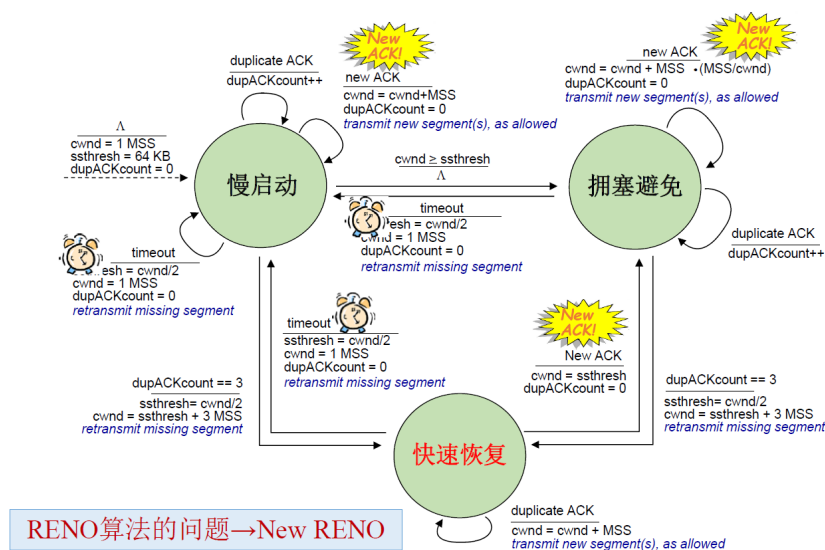


图 3.4: RENO 算法状态机

4 实验环境

本次实验使用 VS2022 进行编程，分为 C/S 两个项目文件。

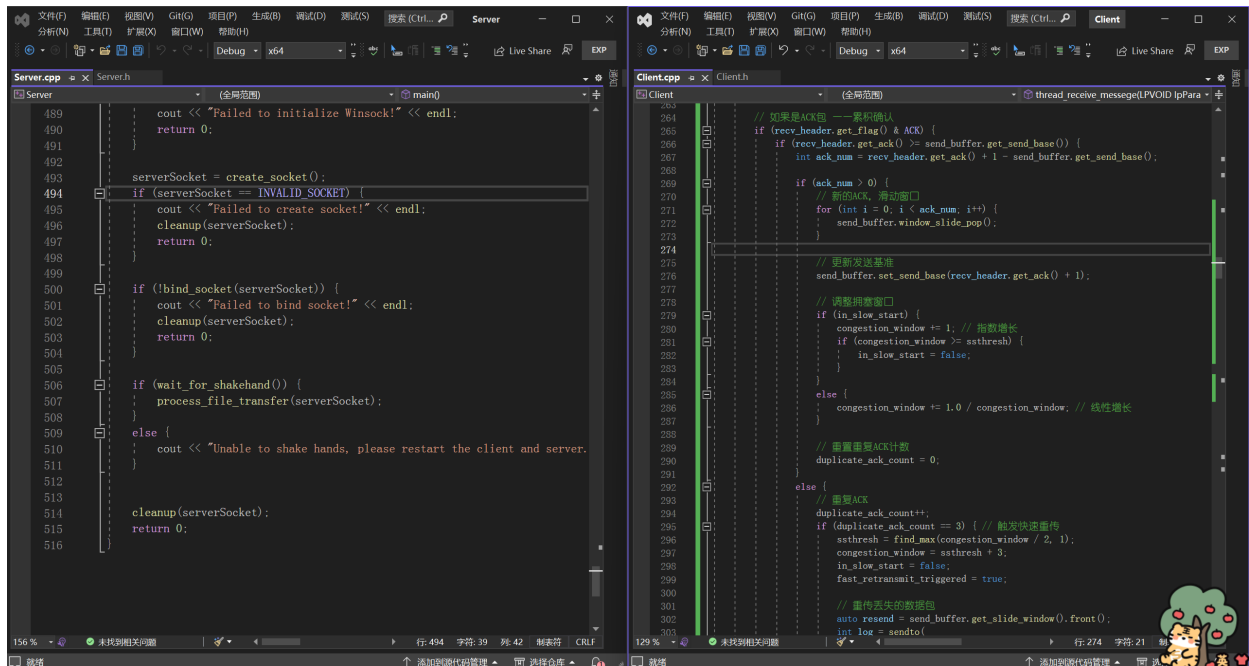


图 4.5: 实验环境

5 UDP 报文设计

如下表所示，是我本次实验所设计的 UDP 报文格式。对应的报文设计与实验 3-1 完全相同，没有做任何修改。

Sequence Number
Acknowledgment Number
Flags
Checksum
Data Length
Header Length
Data

表 1: UDP 报文设计

总体来说，报文由 12 字节的定长报文首部和变长的数据部分组成。其中首部各字段含义如下：

1. **Sequence Number:** 序列号 SEQ，为 16 位，作用是为了避免重传等错误。在正常情况下，我们只需要使用 0 和 1 即可；在挥手过程中，我们采用 256 位的随机序列号，序列号的范围为 0-256。
2. **Acknowledgment Number:** 即 ACK，为 16 位，主要由标志位 Flags 中的 ack 标志位控制，当 Flags 标志位被置为 1 时，ACK 会用来体现最后正确收到的分组的序列号值，由此来替代 NAK。
3. **Flags:** 标志位，我们设计了 SYN、ACK、FIN、LAS 和 RST，作用分别如下：
 - **SYN:** 1, 用来在握手过程中建立连接。

- **ACK: 2**

- 用来控制 ACK 的值是否有效;
- 用来在握手, 挥手包括停等机制时, 其中一方告诉另一方能够正常地收到。

- **FIN: 4**, 用来在挥手过程中客户端告诉服务器端是否结束连接, 开始挥手。

- **LAS: 8**, 发送方在最后一个数据包中加入, 告诉服务器端这就是发送的文件最后一段, 发送文件即将结束, 相当于最后一个发送的数据报。

- **RST: 16**, 当连接需要由于某种原因立即终止时, 一个设备可以发送一个设置了 RST 位的报文。

4. **Checksum**: 即校验和, 我们在前面的实验原理中已经详细介绍了, 此处不再赘述。

5. **Data Length**: 即数据段的长度。

6. **Header Length**: 即报文头的长度, 也就是我们此处设计的头的长度。

7. **Data**: 就是我们需要传输的数据内容。

6 协议设计

在协议部分, 我主要修改了**数据传输部分的协议**, 对于三次握手和二次挥手的协议均没有做太大的修改, 只修改了部分。

6.1 拥塞控制 & 快速重传

这部分是本次实验设计的重点, 我们首先解释一下我们所设计的协议部分:

首先, 设计我们的拥塞窗口, 我们需要在 3-2 的基础上做一些修改, 将我们的窗口大小修改为 **send buffer 和拥塞窗口的最小值**。这样就可以实现拥塞控制的窗口大小调整了。

在慢启动阶段, 我们需要将拥塞窗口的大小初始化为 1, **每当发送方收到一个 ACK, 就会将我们的拥塞窗口的大小翻倍**, 这样就达到了指数增长的要求。

在拥塞避免阶段, 当我们的拥塞窗口增大到一个上限值的时候, 我们就会**调整我们的阶段到拥塞避免阶段**。此时我们的拥塞窗口从指数增长变成了线性增长。

在丢失检测阶段, 首先是超时重传机制。我们首先判断我们设定的时钟有没有超时, 然后将我们的**阈值大小调整到拥塞窗口大小的一半**, 再将**拥塞窗口初始化为 1**。再设定我们的状态为慢启动阶段, 清空重复 ACK 数量即可。

当我们收到 3 个重复的 ACK 的时候, 我们只需要将我们的**快速重传设置为开启**, 这样程序就可以执行我们的快速重传机制了。

对于快速重传的设计, 我们每收到一个新的 ACK (非重复 ACK), 会根据 Reno 算法调整拥塞窗口, 具体也就是**取拥塞窗口增加后的数值与慢启动阈值的最小值**; 当拥塞窗口达到慢启动阈值时, 会退出快速恢复阶段, 进入拥塞避免阶段, 并记录进入拥塞避免阶段的日志。

6.2 三次握手

本部分协议, 3-3 实验与 3-1 实验的协议是一样的, 我没有做对应的修改, 因此不再阐述, 直接将 3-1 部分的协议放在下面:

首先, 当客户端想要与服务器建立连接, 于是客户端向服务器发送 **SYN 报文请求连接**; 服务器收到客户端的连接请求之后, 服务器向客户端发送**确认报文 ACK 及请求连接报文 SYN**; 最后, 客户端收到服务器的连接请求, 向服务器发送**确认报文 ACK**。

6.3 客户端 & 服务端数据传输

该部分基本参照了实验 3-2 的设计思路, 在 3-2 的滑动窗口协议的基础之上进行了拥塞控制的实现。有关拥塞控制与快速重传的设计, 在前面的协议部分已经涉及。有关于滑动窗口的协议, 我在 3-2 已经详细说明, 此处仅仅做一下概括:

我们的接收端的窗口大小需要设置为 1, 而我们的发送端的窗口大小需要设置为 20-32, 所以我们在客户端设计对应的输入发送端窗口大小即可, 在接收端则不需要进行窗口大小的读取, 直接设置为 1 即可。

我们只使用 ACK 来确认按序正确接收的最高序号分组, 这样操作会产生重复的 ACK, **所以我们需要保存希望接收的分组序号 (expectedseqnum)**。

而对于乱序的分组来说, **我们直接不进行缓存, 直接丢弃掉**, 重发 ACK, 确认按序正确接收的最高序号分组即可。

对于数据传输部分的错误处理, 我们在 3-2 中已经具体阐述了, 此处不再赘述。

对于服务器端, 由于我们只需要进行期望序列号的数据包的接收, 而且我们的接收窗口大小为 1, 所以其实我们只需要完成一件事, 就是等待接收目前在接收窗口中的对应序列号的数据包即可。

但是对于客户端, 我们需要同时处理滑动窗口中数据的移入与移出, 如果不控制线程的先后顺序, 会导致很大的错误, 所以在此我们设计了对应的**多线程控制客户端窗口的变化**。

有了多线程, 我对应的加入了**互斥锁的机制**, 解决了争抢资源的问题。

6.4 两次挥手

对于两次挥手部分的协议, 我们保持了与实验 3-2 一致的设计, 没有做出改变。

在第一次挥手中, 客户端向服务器端发送 **FIN=1, seq=nextseqnum 的数据包**, 然后是第二次挥手: 服务器端接收到了客户端的 FIN 报文后, 回复 **ACK=1, ack=nextseqnum+1 的数据包**, 回应对方自己收到了刚才的 FIN 报文。这样我们的挥手就结束了。

7 功能实现 & 代码分析

7.1 Server.h&Server.cpp

本次实验, 我没有对服务器端的代码做出修改, 因为在 3-2 中和 3-3 中都只需要实现接收端窗口的大小为 1 即可, 所以不需要调整我们的服务器端的代码, 此处不再阐述。

7.2 Client.h

本次实验对头文件进行了一些修改, 代码部分如下所示:

```
1 //新增全局变量
2 int congestion_window = 1;           // 拥塞窗口, 初始为 1 个 MSS
3 int ssthresh = 16;                   // 慢启动阈值, 初始值可以设为 16 个 MSS
4 int duplicate_ack_count = 0;         // 重复 ACK 计数
```



```
5 bool fast_retransmit_triggered = false; // 是否触发快速重传
6 bool in_slow_start = true;           // 是否处于慢启动阶段
```

我们在 Client.h 中添加了一些全局变量，用于实现我们的拥塞控制的功能，我们来做一下分析：

- **congestion_window**: 表示拥塞窗口大小，其初始值及后续变化会根据不同阶段和收到的反馈进行调整，以此控制发送数据量的多少来应对网络拥塞情况。
- **ssthresh**: 慢启动阈值，决定何时从慢启动阶段进入拥塞避免阶段，其值会根据网络状况动态更新，例如在超时或收到重复 ACK 等情况时进行相应调整。
- **in_slow_start**: 布尔类型变量，用于标记当前是否处于慢启动阶段，便于执行对应阶段的拥塞窗口调整策略。
- **duplicate_ack_count**: 记录重复 ACK 的数量，当达到 3 次重复 ACK 时，会触发快速重传机制，并且后续会影响拥塞窗口等参数的调整。
- **fast_retransmit_triggered**: 用于标记是否触发了快速重传机制，在相应阶段会依据其状态来执行不同的拥塞窗口调整逻辑。

7.3 Client.cpp

本次主要对客户端的实现做出了修改，我们下面进行详细的分析，**由于分析的篇幅过于冗长，所以我在本次实验报告中，对于与前面实验中相同的部分不再进行阐述，可以详见我之前的报告内容。**

本次修改的部分实现了拥塞控制的全部功能，首先，我们调整了我们发送窗口的大小控制，原本直接使用了滑动窗口的大小，在此处我们将其修改为滑动窗口和拥塞窗口大小的最小值：

```
1 while (send_buffer.get_next_seq_num() >= send_buffer.get_send_base() +
   ↪ find_min(congestion_window, send_buffer_size)) {
2     std::this_thread::sleep_for(std::chrono::milliseconds(10));
3 }
```

剩下的修改均在**接收线程**中进行实现，逐一分析：

首先，我们实现了**超时重传**的功能，代码如下所示：

```
1 if (client_timer.is_timeout()) {
2     // 超时就重传在窗口内的所有的数据包
3     // 设置慢启动阈值
4     ssthresh = find_max(congestion_window / 2, 1);
5     congestion_window = 1;
6     in_slow_start = true;
7     duplicate_ack_count = 0;
8
9     // 记录重传相关日志，方便查看网络状态变化
10    {
11        lock_guard<mutex> log_queue_lock(log_queue_mutex);
```

```
12     log_queue.push_back("Timeout occurred, entering slow start phase and resent  
    ↪ all datagrams in the window.\n");  
13     log_queue.push_back("New ssthresh: " + to_string(ssthresh) + ", New  
    ↪ congestion_window: " + to_string(congestion_window) + "\n");  
14 }  
15  
16 // 重传窗口内所有数据包  
17 for (auto resend : send_buffer.get_slide_window()) {  
18     int log = sendto(  
19         clientSocket,  
20         (char*)resend,  
21         resend->header.get_data_length() + resend->header.get_header_length(),  
22         0,  
23         (sockaddr*)&serverAddr,  
24         sizeof(sockaddr_in)  
25     );  
26  
27     if (log == SOCKET_ERROR) {  
28         std::cerr << "Failed to resend data packet! Error: " <<  
29             ↪ GetLastErrorDetails() << std::endl;  
30         // 进行相应的错误处理，如重试或退出  
31     }  
32  
33     // 记录重传操作日志  
34     {  
35         lock_guard<mutex> log_queue_lock(log_queue_mutex);  
36         log_queue.push_back("Timeout, resent datagram to server.\n");  
37     }  
38  
39     client_timer.start(); // 重置定时器开始计时  
40 }
```

我们通过 `client_timer` 来判定是否出现了超时的现象，如果超时的话，相当于进入了我们这个循环。首先，我们设置我们的慢启动阈值为 `ssthresh`（取 `congestion_window / 2` 与 `1` 中的较大值），将拥塞窗口 `congestion_window` 重置为 `1`，并标记进入慢启动阶段 `in_slow_start` 设为 `true`），同时重置重复 ACK 计数。然后重传窗口内所有数据包（遍历发送缓冲区的滑动窗口中的数据包进行重传），并记录重传相关日志，包括网络状态变化、新的阈值与拥塞窗口值等。最后重置定时器开始计时，以便后续继续监控数据传输过程中的超时情况。

然后是我们对于慢启动阶段和拥塞避免阶段的处理：

```
1 //定义全局计数器，记录当前收到 ACK 的数量  
2 int ack_num_time = 0;
```

```
3 // 调整拥塞窗口
4 if (in_slow_start) {
5     congestion_window = congestion_window + 1; // 指数增长
6     if (congestion_window >= ssthresh) {
7         in_slow_start = false;
8         lock_guard<mutex> log_queue_lock(log_queue_mutex);
9         log_queue.push_back("Exiting slow start phase, entering congestion
    ↳ avoidance phase.\n");
10    }
11 }
12 else {
13     if (ack_num_time == congestion_window) {
14         congestion_window = congestion_window + 1;
15         ack_num_time = 0; // 重新开始计时
16     }
17
18     lock_guard<mutex> log_queue_lock(log_queue_mutex);
19     log_queue.push_back("In congestion avoidance phase, updated congestion_window:
    ↳ " + to_string(congestion_window) + "\n");
20 }
```

在慢启动阶段，每当收到新的 ACK（基于累积确认机制，新的 ACK 意味着对应数据已被正确接收），拥塞窗口大小按指数增长，也就是代码中的 `congestion_window = congestion_window + 1`；持续增长的拥塞窗口会与慢启动阈值 `ssthresh` 进行比较，当 `congestion_window ≥ ssthresh` 时，将会切换出慢启动阶段，进入拥塞避免阶段，同时记录相应日志提示网络状态发生变化。

进入拥塞避免阶段后，拥塞窗口的增长变为线性方式，为了保证我们的拥塞窗口的大小均为整数，我们定义一个全局变量 `ack_num_time`，每收到一次 ACK 的时候，我们就对其进行加一操作。在拥塞避免阶段中，通过判断 `ack_num_time` 的值是否与拥塞窗口 `congestion_window` 相等，来对其进行加一操作——如果与拥塞窗口大小相等，说明我们此时应该去完成我们的拥塞窗口加一的操作；如果比拥塞窗口要小的话，就不用进行操作。通过这种方式更平缓地调整发送速率，避免网络出现过度拥塞情况，同时会记录日志展示拥塞窗口的更新情况。

然后是我们对于快速重传和快速恢复的处理，代码如下所示：

```
1 else {
2     // 接收到的是重复 ACK
3     duplicate_ack_count++;
4
5     // 判断重复 ACK 的数量，达到 3 次则触发快速重传机制
6     if (duplicate_ack_count == 3) {
7         // 设置慢启动阈值，取当前拥塞窗口的一半与 1 中的较大值
8         ssthresh = find_max(congestion_window / 2, 1);
9         // 调整拥塞窗口大小，依据快速重传策略设置
10        congestion_window = ssthresh + 3;
```

```
11     in_slow_start = false;
12     fast_retransmit_triggered = true;
13
14     // 重传疑似丢失的数据包, 取发送缓冲区滑动窗口的头部数据包进行重传
15     auto resend = send_buffer.get_slide_window().front();
16     int log = sendto(
17         clientSocket,
18         (char*)resend,
19         resend->header.get_data_length() + resend->header.get_header_length(),
20         0,
21         (sockaddr*)&serverAddr,
22         sizeof(sockaddr_in)
23     );
24
25     if (log == SOCKET_ERROR) {
26         std::cerr << "Failed to resend data packet during Fast Retransmit!
27         ↪ Error: " << GetLastErrorDetails() << std::endl;
28         // 提示进行相应错误处理, 例如重试或直接退出程序
29         // 此处可按需添加具体错误处理逻辑
30     }
31
32     // 记录快速重传操作及相关网络状态变化日志
33     {
34         lock_guard<mutex> log_queue_lock(log_queue_mutex);
35         log_queue.push_back("Fast Retransmit: Resent datagram due to triple
36         ↪ duplicate ACKs.\n");
37         log_queue.push_back("Updated ssthresh: " + to_string(ssthresh) + ",
38         ↪ Updated congestion_window: " + to_string(congestion_window) +
39         ↪ "\n");
40     }
41 }
```

当接收到的 ACK 为重复 ACK 时, 代码会对重复 ACK 的数量进行计数(通过 `duplicate_ack_count` 变量), 一旦重复 ACK 的数量达到 3 次, 就会触发快速重传机制。

触发快速重传时, 会对多个关键参数进行调整, 具体如下:

- `ssthresh` 调整为当前 `congestion_window/2` 与 1 中的较大值。
- `congestion_window` 设置为 `ssthresh + 3`。
- 同时标记进入快速恢复阶段 (`fast_retransmit_triggered` 设为 `true`), 然后重传丢失的数据包 (依据相应逻辑从发送缓冲区找到对应数据包并重新发送), 并且记录重传操作及相关状态变化日志。

然后我们再来分析我们对于快速重传, 也就是 `fast_retransmit_triggered` 为 `true` 时的设计:

```
1 // 判断是否已触发快速重传机制
2 if (fast_retransmit_triggered) {
3     // 处于快速恢复阶段，检查接收到的 ACK
4     if (recv_header.get_ack() > send_buffer.get_send_base()) {
5         // 根据 Reno 算法调整拥塞窗口，此处原代码简化处理，可优化计算增加量
6         congestion_window = find_min(congestion_window + 1, ssthresh);
7
8         // 如果拥塞窗口达到慢启动阈值，切换进入拥塞避免阶段
9         if (congestion_window == ssthresh) {
10             in_slow_start = false;
11             fast_retransmit_triggered = false;
12
13             // 记录进入拥塞避免阶段的日志信息
14             {
15                 lock_guard<mutex> log_queue_lock(log_queue_mutex);
16                 log_queue.push_back("Exiting fast recovery phase, entering
17                                     ↪ congestion avoidance phase.\n");
18             }
19         }
20     }
```

我们首先需要修改我们的发送窗口的大小，也就是前面的协议设计中所提到的拥塞窗口增加后的数值与慢启动阈值的最小值，如果拥塞窗口的大小达到了慢启动的阈值，那么我们就切换我们进入到拥塞避免的阶段，设置慢启动状态为 false，快速重传状态也为 false。同时，记录下我们当前的日志信息。

8 结果展示

8.1 无丢包测试

首先，我们测试一下丢包率为 0 的情况下，我们的传输效果。

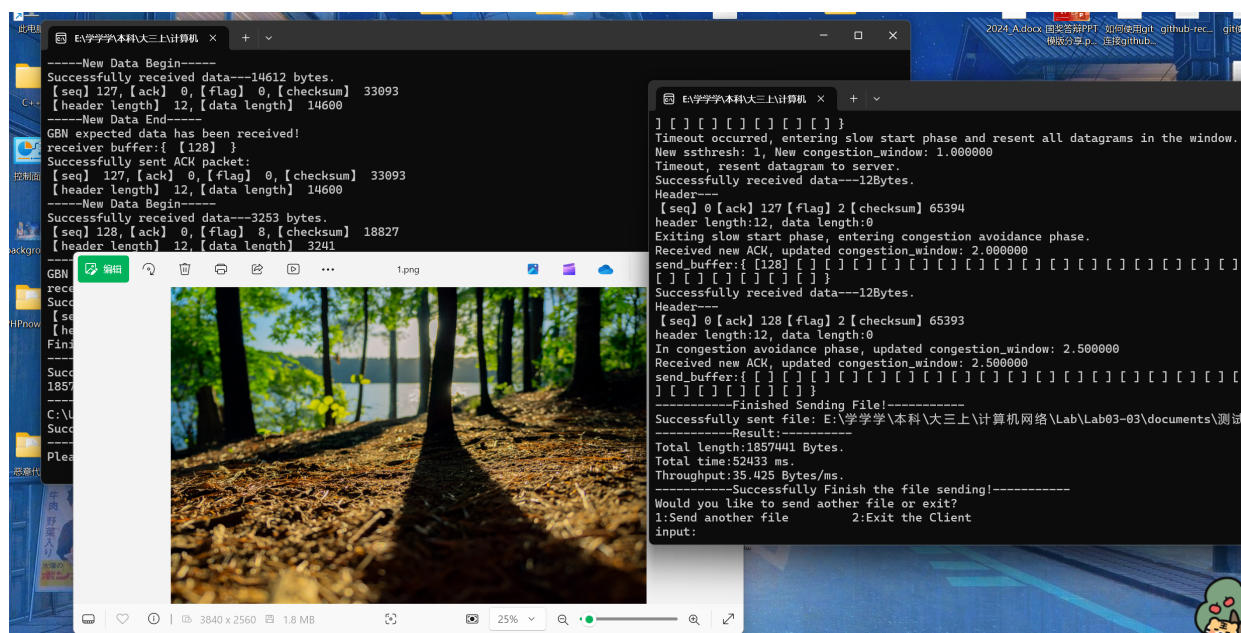


图 8.6: 无丢包

我们发现，在添加了拥塞窗口之后，我们也成功完成了我们的数据传输的任务。

8.2 有丢包测试

我们再来对有丢包的情况进行分析：

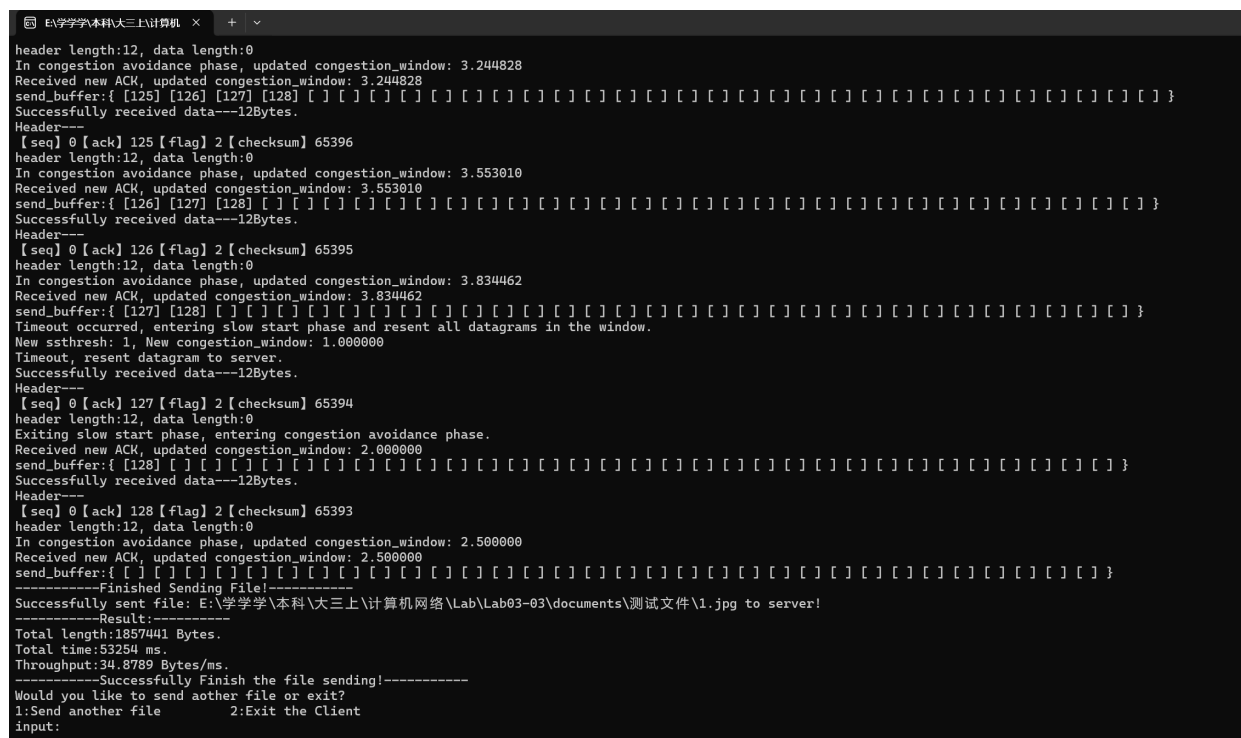


图 8.7: 有丢包测试

我们发现，在传输的过程中，出现了拥塞避免、快速重传、慢启动等过程，说明我们的设计是正确

的，测试完毕。

我还发现，当我们的丢包率比较小的时候，我们基本上不会遇到超时重传的机会，当我们把丢包率调整为 90% 的时候，传输过程如下所示：

```

send_buffer:{ [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26]
34] [35] [36] [37] [38] [39] [ ] [ ] [ ] [ ] [ ] [ ] }
-----simulate to drop the data for server!-----
-----simulate to drop the data for server!-----
-----simulate to drop the data for server!-----
-----simulate to drop the data for server!-----
-----simulate to drop the data for server!-----
Successfully received data---12Bytes.
Header---
【seq】 0 【ack】 12 【flag】 2 【checksum】 65509
header length:12, data length:0
-----RELATIVE DELAY TIME!-----
Timeout occurred, entering slow start phase and resent all datagrams in the window.
New ssthresh: 1, New congestion_window: 1.000000
Timeout, resent datagram to server.
Successfully received data---12Bytes.
Header---
【seq】 0 【ack】 13 【flag】 2 【checksum】 65508
header length:12, data length:0
Exiting slow start phase, entering congestion avoidance phase.
Received new ACK, updated congestion_window: 2.000000
send_buffer:{ [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27]

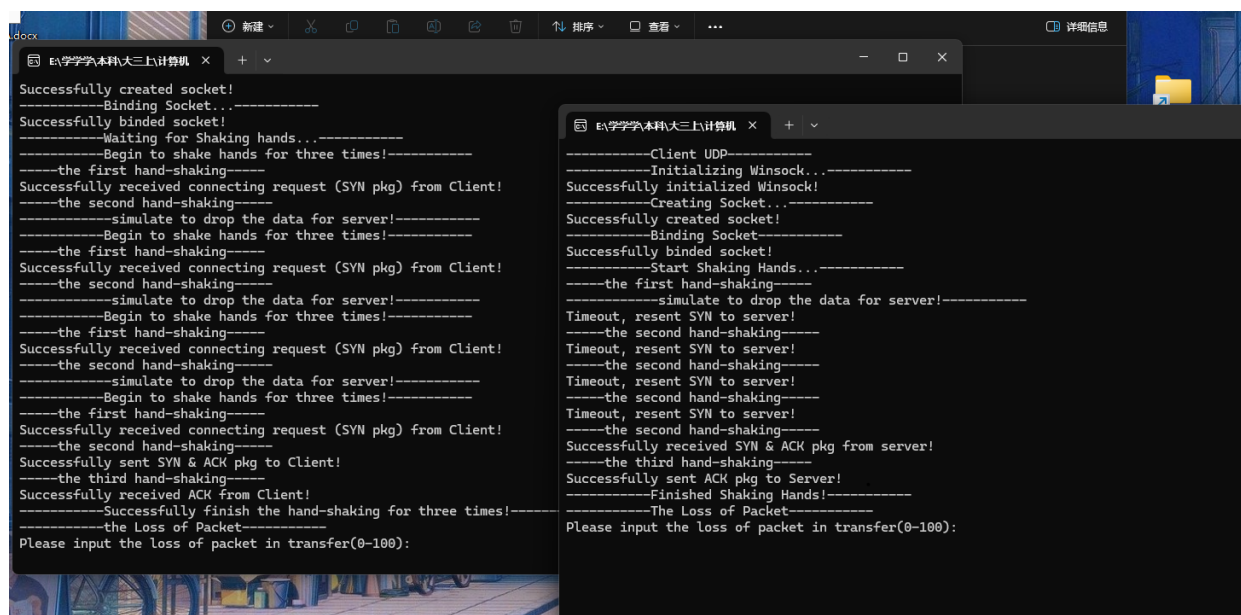
```

图 8.8: 超时重传

我们发现，出现了对应的超时重传的情况。

8.3 三次握手测试

我们来进行起始的三次握手的测试：



```

Client:
-----Client UDP-----
-----Initializing Winsock...-----
Successfully initialized Winsock!
-----Creating Socket-----
Successfully created socket!
-----Binding Socket-----
Successfully binded socket!
-----Start Shaking Hands-----
-----the first hand-shaking-----
-----simulate to drop the data for server!-----
Timeout, resent SYN to server!
-----the second hand-shaking-----
Timeout, resent SYN to server!
-----the second hand-shaking-----
Timeout, resent SYN to server!
-----the second hand-shaking-----
Timeout, resent SYN to server!
-----the second hand-shaking-----
Successfully received SYN & ACK pkg from server!
-----the third hand-shaking-----
Successfully sent ACK pkg to Server!
-----Finished Shaking Hands!-----
-----The Loss of Packet-----
Please input the loss of packet in transfer(0-100):

Server:
Successfully created socket!
-----Binding Socket-----
Successfully binded socket!
-----Waiting for Shaking hands...-----
-----Begin to shake hands for three times!-----
-----the first hand-shaking-----
Successfully received connecting request (SYN pkg) from Client!
-----the second hand-shaking-----
-----simulate to drop the data for server!-----
-----Begin to shake hands for three times!-----
-----the first hand-shaking-----
Successfully received connecting request (SYN pkg) from Client!
-----the second hand-shaking-----
-----simulate to drop the data for server!-----
-----Begin to shake hands for three times!-----
-----the first hand-shaking-----
Successfully received connecting request (SYN pkg) from Client!
-----the second hand-shaking-----
-----simulate to drop the data for server!-----
-----Begin to shake hands for three times!-----
-----the first hand-shaking-----
Successfully received connecting request (SYN pkg) from Client!
-----the second hand-shaking-----
Successfully sent SYN & ACK pkg to Client!
-----the third hand-shaking-----
Successfully received ACK from Client!
-----Successfully finish the hand-shaking for three times!-----
-----the Loss of Packet-----
Please input the loss of packet in transfer(0-100):

```

图 8.9: 三次握手测试

我们发现，三次握手的三个阶段均成功，测试完毕！

8.4 两次挥手测试

然后在文件传输完毕后，我们尝试进行两次挥手来结束我们的进程，如下所示：

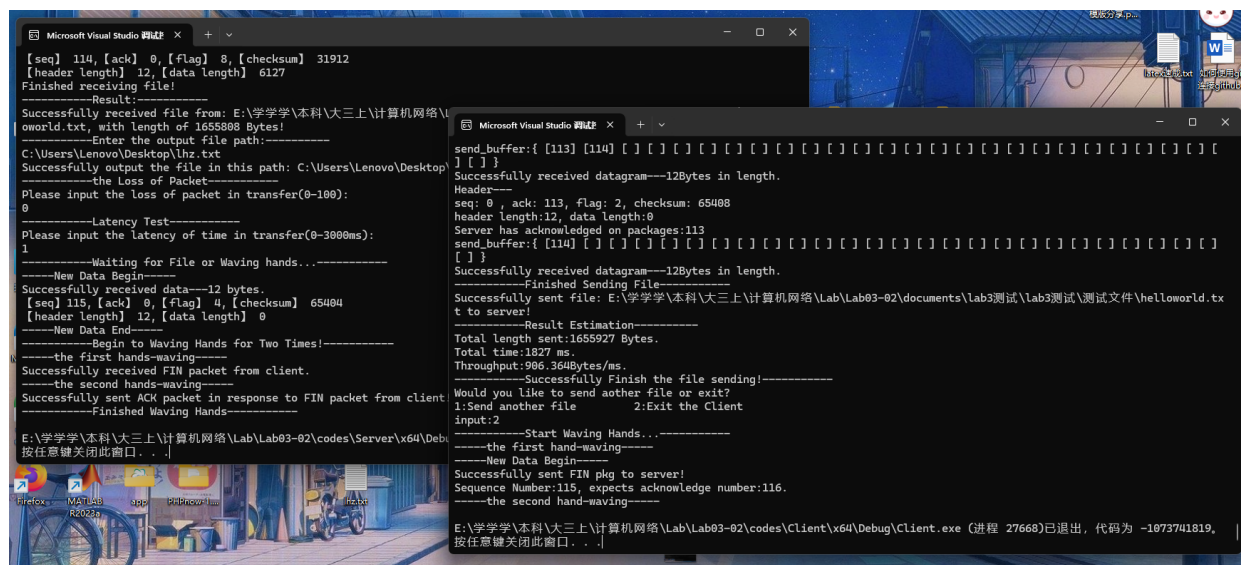


图 8.10: 两次挥手测试

我们发现，程序成功地完成了两次挥手，并且均退出了进程。测试完毕！

9 心得体会

本次实验在实验 3-2 的基础上，进一步实现了拥塞控制，在丢包率较大的时候，提升了文件的传输效率。当然，在前三次的实验中，我已经基本熟悉了这几种实现方式的优缺点，在下一 3-4 的实验中，我将会把这几种方法做一个对比，通过控制变量对比测试来进一步深入了解和思考这几种机制的优缺点。