实验 3-4 性能对比

1813800 沈 哲

目录

1 实验要求 **1**

2 性能对比与分析 **1**

2.1 停等机制与滑动窗口机制 1

2.2 滑动窗口机制的不同窗口大小 4

2.3 有拥塞控制和无拥塞控制 6

3 附录 1：实验 3-2、3-3 的功能实现 **9**

3.1 协议设计 9

3.2 建立与断开连接 9

3.2.1 建立连接 9

3.2.2 断开连接 10

3.3 差错检测 10

3.4 确认重传 10

3.5 基于滑动窗口的流量控制机制 10

3.5.1 发送端 11

3.5.2 接收端 11

3.6 拥塞控制（RENO 算法） 12

3.6.1 慢启动 12

3.6.2 拥塞避免 12

3.6.3 快速恢复 12

3.6.4 三种状态的转换 12

3.7 其他设计 13

3.7.1 定时器的设计 13

3.7.2 随机丢包和延时的设计 13

3.7.3 快速重传和窗口加速滑动 14

3.7.4 测试方法 14

4 附录 2：实验 3-2、3-3 的代码分析 14
4.1 公共头文件定义 . . . 14
4.2 基本函数（不传入参数，直接执行完成相应功能） . . . 15
4.2.1 makePackage . . . 15
4.2.2 doCheckSum . . . 15
4.2.3 Check_expected_ACK(发送端使用) . . . 16
4.2.4 CheckSEQ_EXPECTED(接收端使用) . . . 16
4.2.5 IsACK,IsFIN,IsSYN . . . 16
4.3 滑动窗口的发送端代码 . . . 16
4.3.1 窗口滑动 . . . 17
4.3.2 接收 ACK . . . 18
4.3.3 超时重传 . . . 19
4.4 滑动窗口的接收端代码 . . . 20
4.4.1 接收按序的分组并写入文件 . . . 20
4.4.2 接收到失序的分组, 直接丢弃 . . . 21
4.5 拥塞控制代码——在发送端进行三种状态的转换 . . . 21
4.6 断开连接的其他辅助处理 . . . 22
4.6.1 发送端可能收不到接收端回复的 FIN . . . 22

5 附录 3：实验 3-2、3-3 的结果展示 23
5.1 实验 3-2 结果展示 . . . 23
5.2 实验 3-3 结果展示 . . . 25

1 实验要求

基于给定的实验测试环境，通过改变延迟时间和丢包率，完成下面 3 组性能对比实验：

（1）停等机制与滑动窗口机制性能对比；

（2）滑动窗口机制中不同窗口大小对性能的影响；

（3）有拥塞控制和无拥塞控制的性能比较。

要求实现单向传输。对于每一个任务要求给出详细的协议设计。完成给定测试文件的传输，显示传输时间和平均吞吐率。性能测试指标包括吞吐率和时延，给出图形结果并进行分析。

2 性能对比与分析

本次实验只需在前三次实验基础上进行性能测试，前三次代码的架构已经修改为一致，便于测试。前三次实验的内容可以参考附录 (3)。

吞吐率是即单位时间内实际传输的数据量，时延从网络中的一端开始发送数据算起，到另一端完全接收这个数据的时间间隔。都采用多次测量取平均值的方法。

实验中设置的数据缓冲区大小是 65310 字节。

2.1 停等机制与滑动窗口机制

1. 延迟时间 200ms，丢包率 30%。

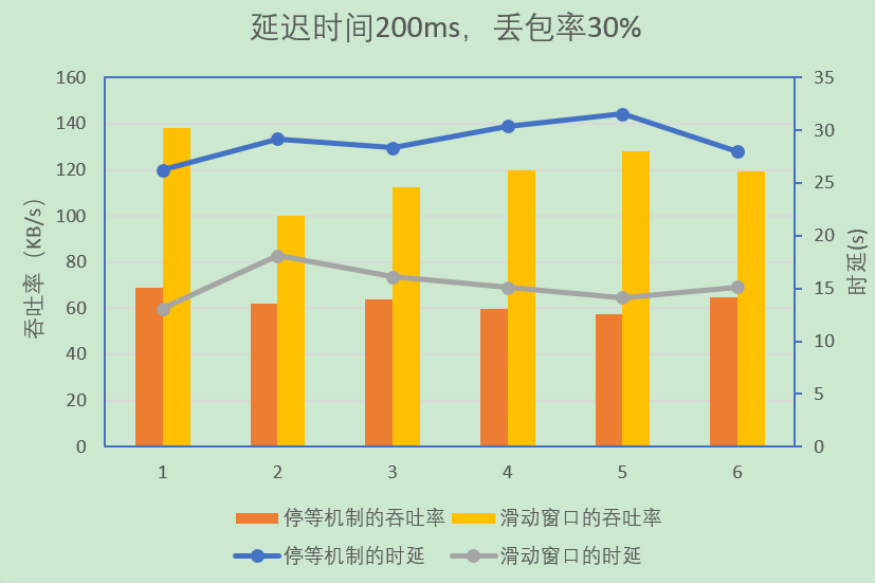


Fig. 1: 延迟时间 200ms，丢包率 30%

2. 延迟时间 200ms，丢包率 50%。

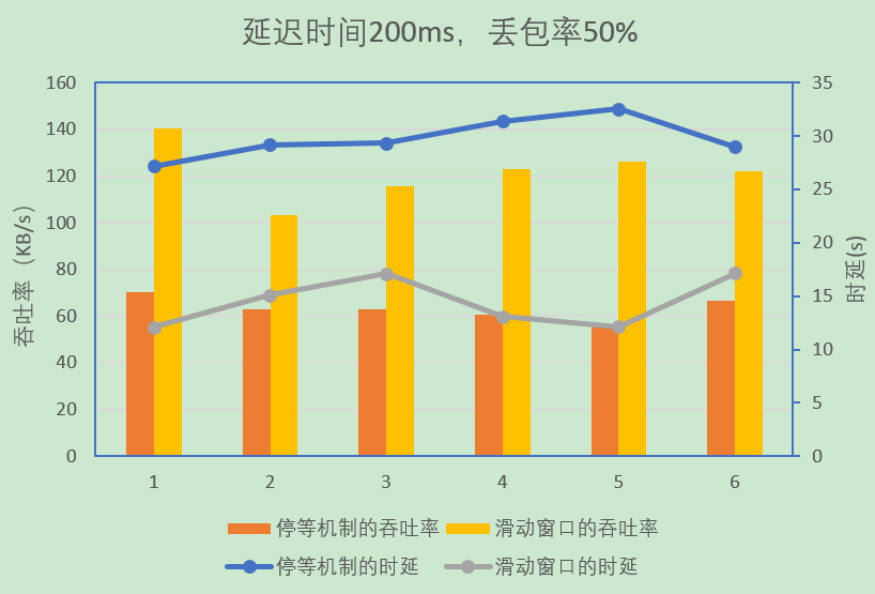


Fig. 2: 延迟时间 200ms，丢包率 50%

3. 延迟时间 500ms，丢包率 30%。

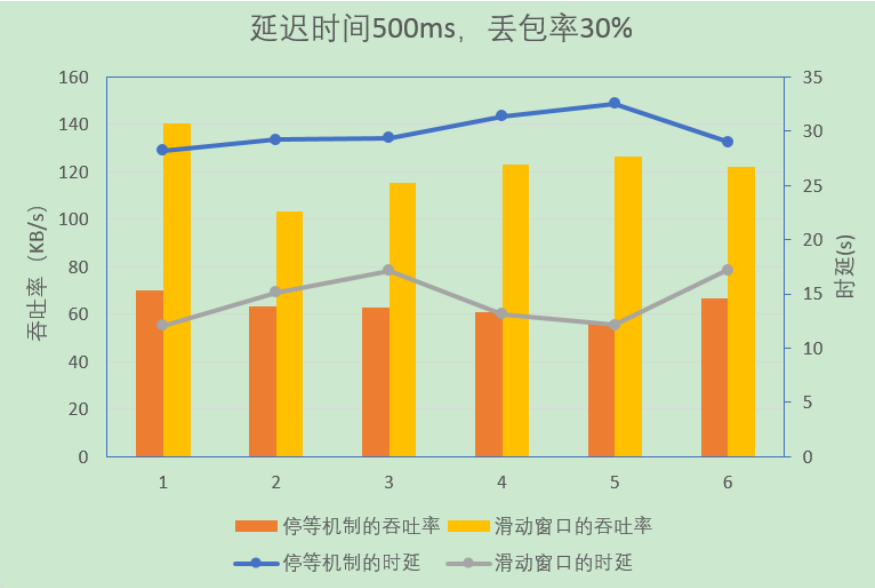


Fig. 3: 延迟时间 500ms，丢包率 30%

4. 延迟时间 500ms，丢包率 50%。

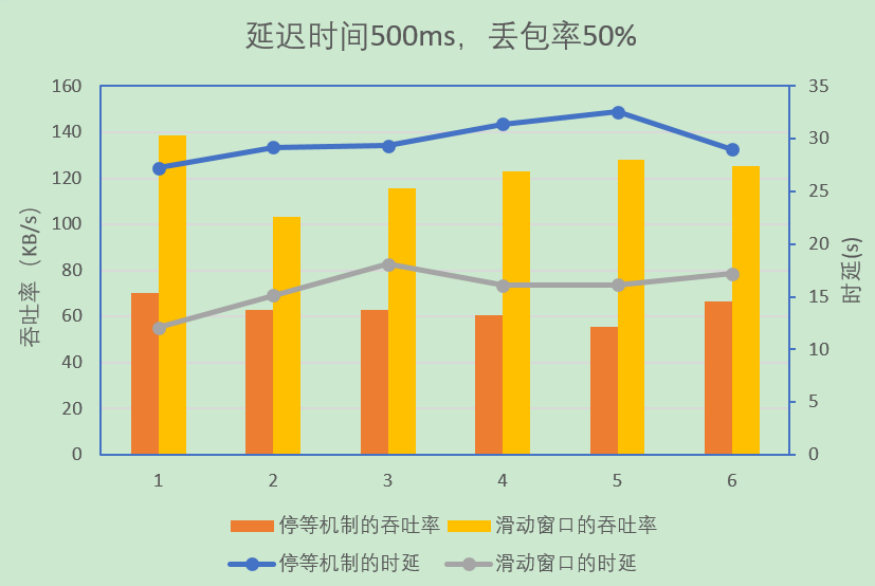


Fig. 4: 延迟时间 500ms，丢包率 50%

通过四组对比可以明显看出，滑动窗口机制比停等机制的传输时延更短，吞吐率更高。

停等机制可以看作窗口大小为 1 的滑动窗口机制，每次发送分组都要得到确认后才能继续发送下一个。滑动窗口机制一次发送多个分组，传输效率更高，即使遇到重传情况，也能获得更大的吞吐率和更小的时延。

2.2 滑动窗口机制的不同窗口大小

1. 延迟时间 300ms，丢包率 30%。

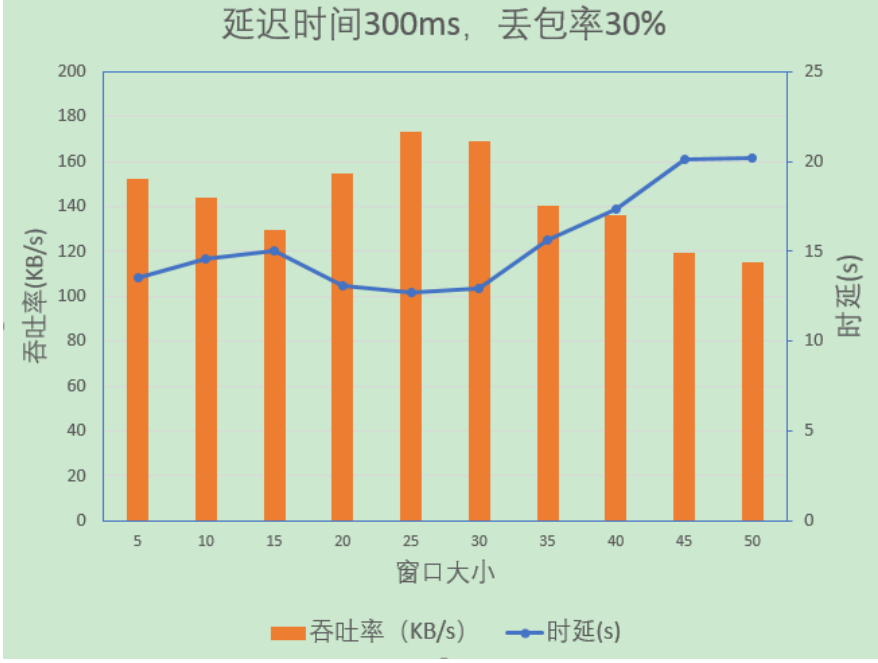


Fig. 5: 延迟时间 300ms，丢包率 30%

2. 延迟时间 300ms，丢包率 50%。

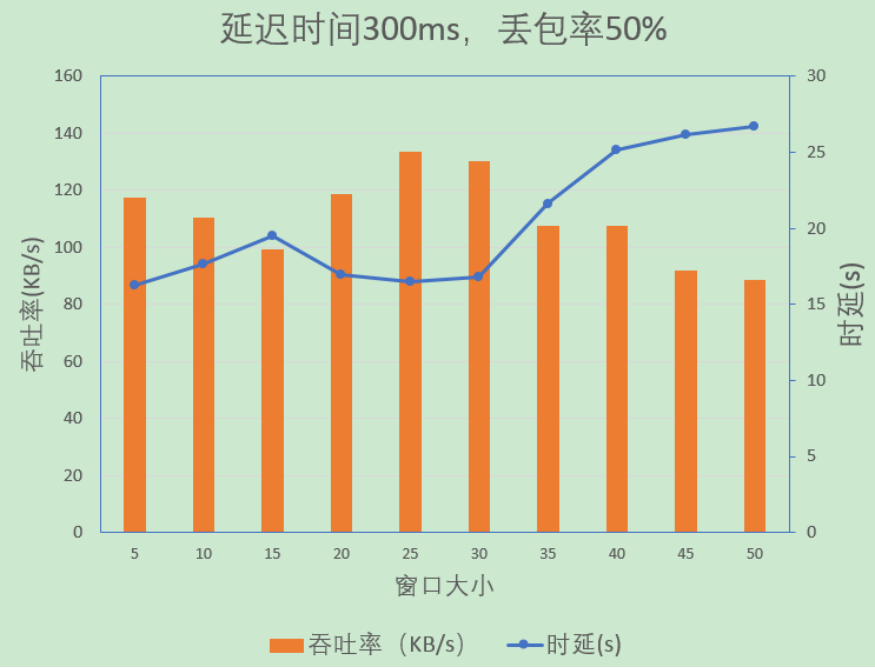


Fig. 6: 延迟时间 300ms，丢包率 50%

3. 延迟时间 600ms，丢包率 30%。

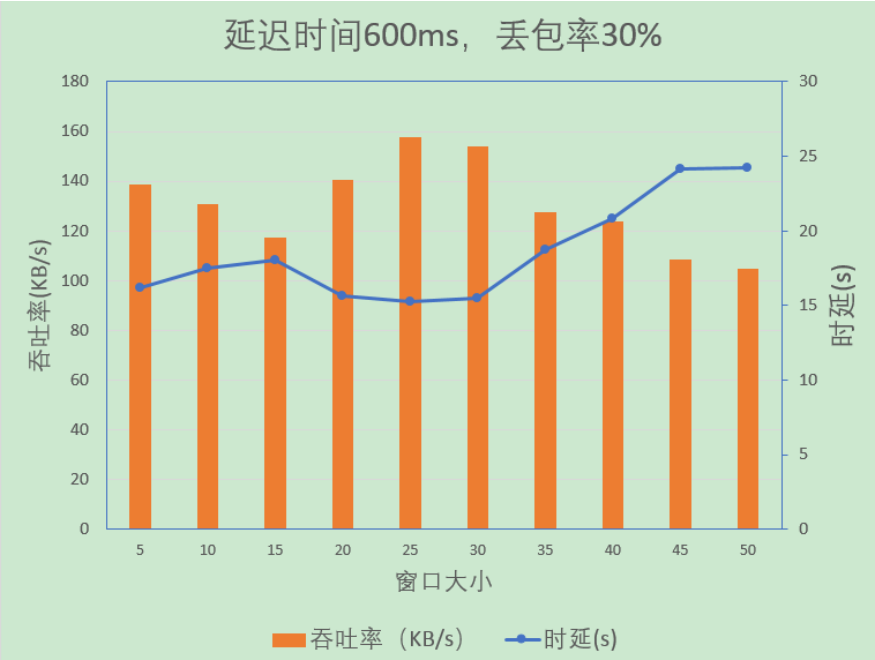


Fig. 7: 延迟时间 600ms，丢包率 30%

4. 延迟时间 600ms，丢包率 50%。

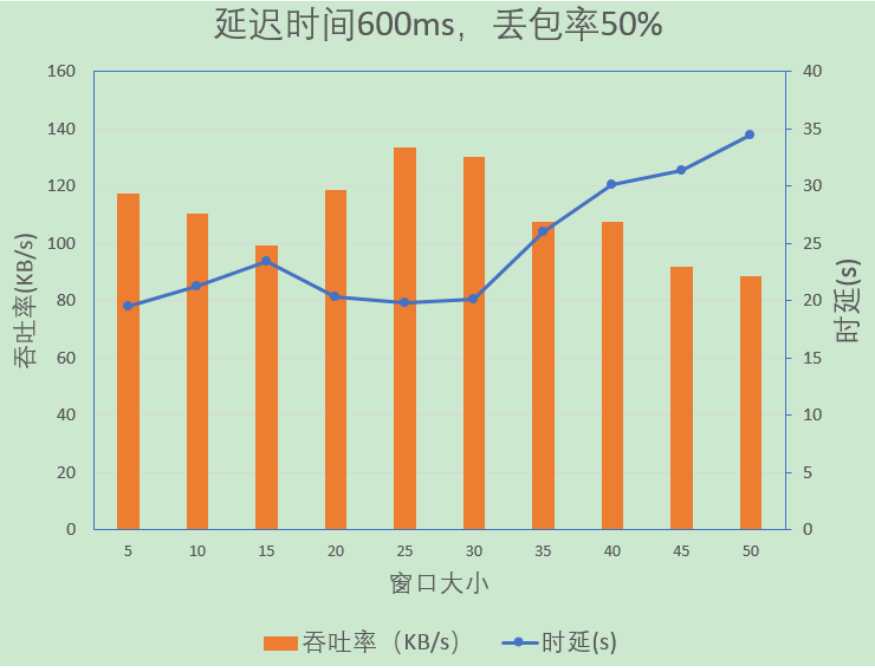


Fig. 8: 延迟时间 600ms，丢包率 50%

通过图形结果可以看出:

丢包率不变时，增大延迟时间，传输的时延增大，吞吐率降低。因为吞吐量不变的情况下传输等待时间变长。

延迟时间不变时，增大丢包率，传输的时延增大，吞吐率降低。因为更多的丢包会造成更多的分组重传。

窗口大小在 20-30 之间时可以获得较大的吞吐率和较低的时延。

窗口较小时（5-20），随着窗口增大，吞吐率降低，猜测应该是因为随着窗口变大，重传的代价变大。这种情况下较小的窗口重传花费时间更短，时延更低。

但随着窗口继续增大（20-30），窗口增大带来的增益超过重传产生的时延代价，即较大的窗口能够传输更多的数据量，即使偶尔重传，造成的时延也可以抵消，可以获得最好的效果。

当窗口变得更大（30-50），窗口增大的优势变得不明显，而重传带来的时延损失变得严重，导致最终的时延增大，吞吐率降低。

2.3 有拥塞控制和无拥塞控制

1. 延迟时间 300ms，丢包率 30%。

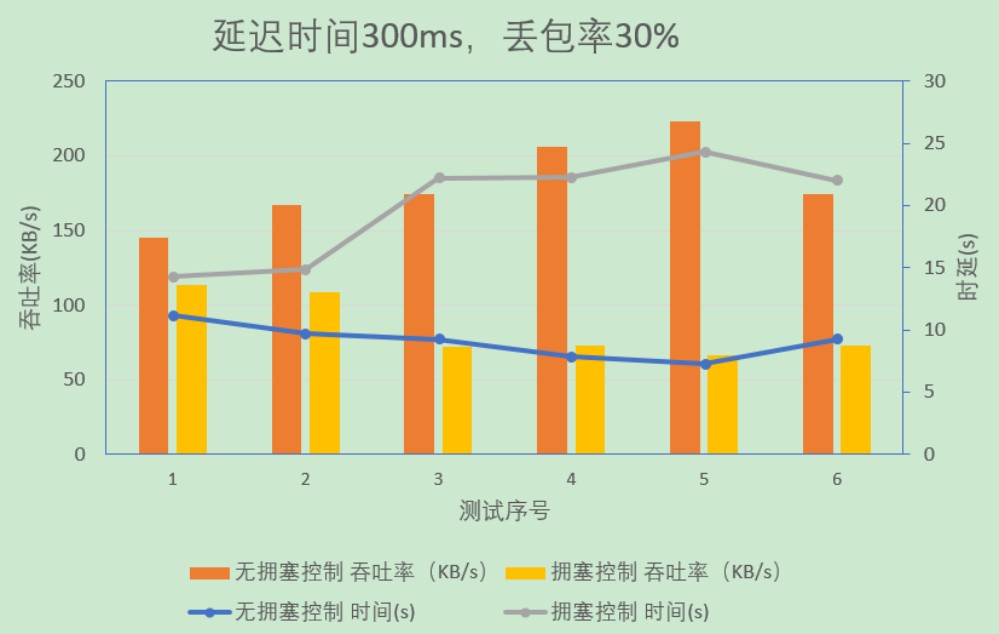


Fig. 9: 延迟时间 300ms，丢包率 30%

2. 延迟时间 300ms，丢包率 50%。

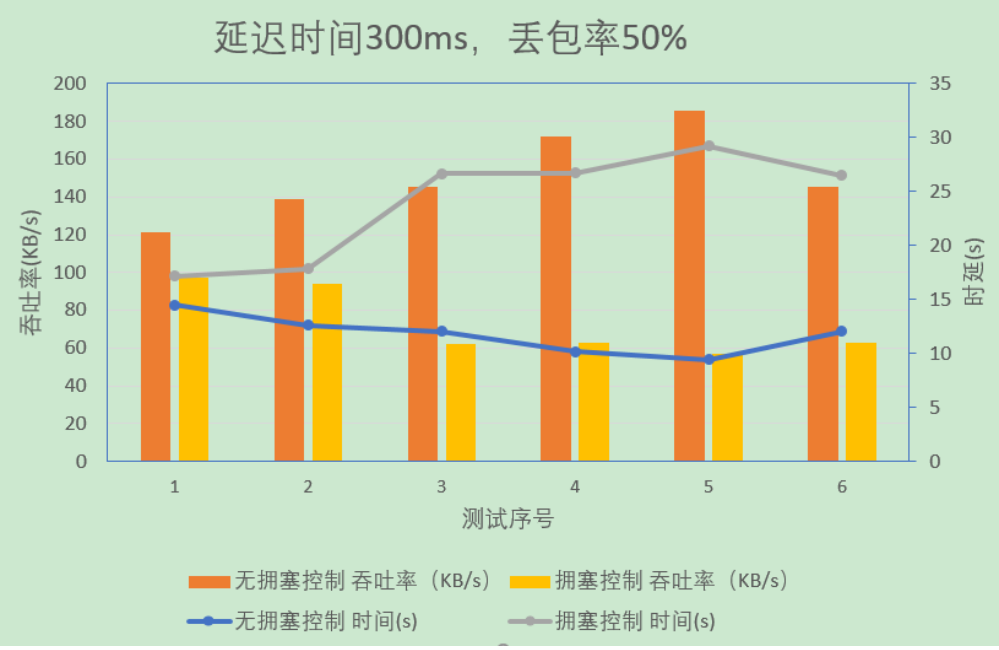


Fig. 10: 延迟时间 300ms，丢包率 50%

3. 延迟时间 600ms，丢包率 30%。

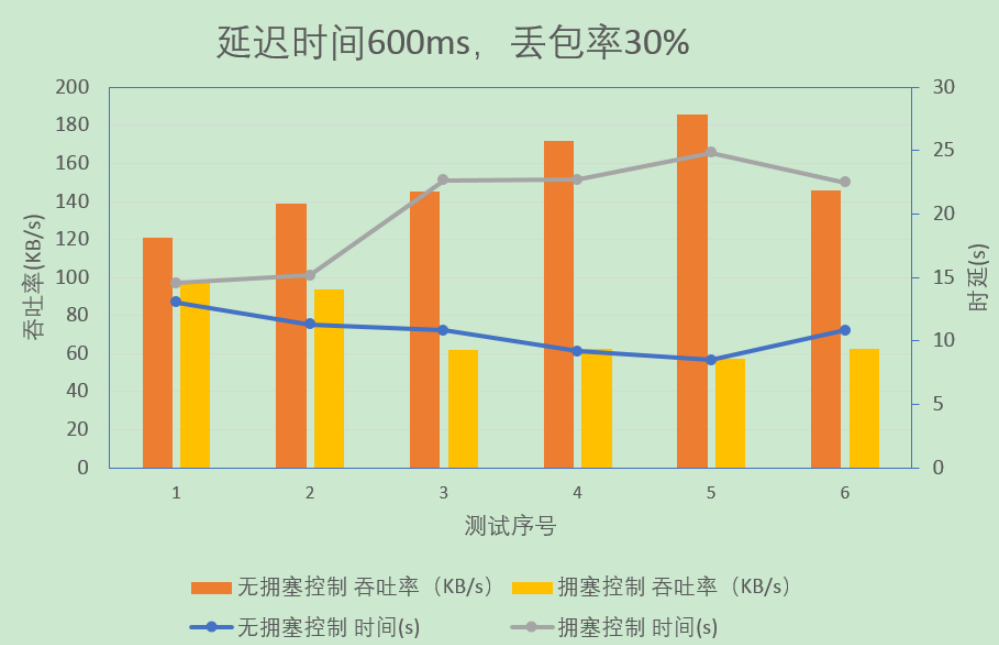


Fig. 11: 延迟时间 600ms，丢包率 30%

4. 延迟时间 600ms，丢包率 50%。

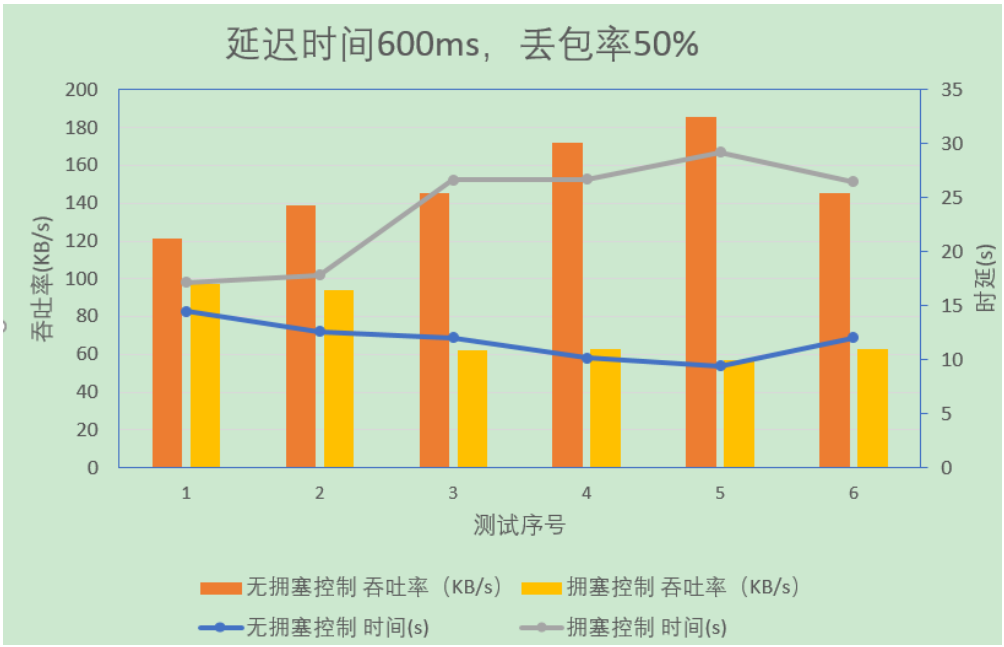


Fig. 12: 延迟时间 600ms，丢包率 50%

由于无法正常使用路由程序测试，本实验的丢包处理和延时处理是在程序中写入的，丢包使用随机数方法，延时使用 Sleep 函数。所以可能无法很好模拟网络拥塞环境。

在前几组序号的测试中，可以看到拥塞控制版本的吞吐率和滑动窗口版本接近，这是因为滑动窗口较小，拥塞控制能够取得相近的效果。但当滑动窗口增大到合适后，无拥塞控制的滑动窗口版本吞吐率上升很快，拥塞控制无法达到这样的效果。

测试结果显示拥塞控制的效果不如滑动窗口效果。原因可能是：

在无拥塞控制的滑动窗口机制中窗口的优势更容易显现，但在拥塞控制中，由于丢包原因很容易造成超时事件，就会多次进入慢启动状态，窗口变为 1, 再重新进入增大的过程，导致吞吐率降低。

即使进入拥塞避免和快速恢复，窗口增加带来的效果也比不上滑动窗口固定窗口大小的效果。并且这两种状态很容易再次转移到慢启动状态。而固定窗口大小能够一直保持稳定的传输速率，获得较低的时延和较高的吞吐率。

本实验无法使用路由程序，难以模拟真实网络环境，所以无法观测拥塞控制的真正效果。本实验中的丢包只是随机丢包并不能代表网络拥塞状况，所以拥塞控制做出的反应并不能有效提升反而降低了吞吐率。猜想如果在真实网络环境中，有拥塞控制会比无拥塞控制的机制取得更好的效果。考虑之后创造条件进行更完善的测试。

3 附录 1：实验 3-2、3-3 的功能实现

3.1 协议设计

UDP 是传输层中面向无连接的协议，在编程上服务端和客户端是没有区别的，本实验实现从客户端（发送端）到服务端（接收端）的传输。本实验参考了 TCP 协议来设计数据报相应字段，数据报分为两部分——头部和数据部分。

实验 3-3 的协议和 3-2 完全相同，接近 TCP，数据报的结构更容易处理（使用结构体而不是字符数组，虽然字符数组更简单更易于传送，但编程操作上可能更加困难）。结构体 PACKAGE 仍是 10 字节的头部 + 数据部分，占用字节数不变，但操作变得简单，可以直接对结构体属性赋值。属性类型全部改用 unsigned short, 表示 0-65535 范围的无符号整数。

TCP 是字节流传送（序列号每次增加的值为字节数），本协议简单地使用序列号递增模式。即 SEQ 每次递增 1（等于接收到的 ACKnum，可以对 65535 取模），ACKnum 是对方序列号加 1（表示期待对方下次发送的序列号）：send_SEQ=recv_ACKnum,send_ACKnum=recv_SEQ。本实验中 SEQ 主要在发送端使用，ACKnum 主要在接收端使用。

数据部分占 65300 字节。头部包括：序列号、确认序号、检验和字段、标志位字段、长度字段。每一部分占 2 字节，共占 10 字节。

```
Header:10 bytes
| 2 bytes  | 2 bytes  | 2 bytes  | 2 bytes  |  2 bytes  |
|   SEQ    |  ACKnum  | CheckSum |    ID  |   Length  |
```

SEQ 是序列号，范围是 0 到 65535，主要在发送端使用。

ACKnum 是确认序列号，主要在接收端使用，接收端回复的 ACKnum 等于发送端的 SEQ+1,（代表下次期望收到的发送端序列号）。

CheckSum 是检验和字段，发送端制作分组时计算检验和填入，接收端对收到的分组进行差错检验。

ID 是多位标志位，包括 SYN（连接建立标志位）、ACK（确认标志位）、FIN（断开连接标志位）。

```
ID:2 bytes
|  FIN  |  ACK  |  SYN  |
```

Length 是数据长度。

3.2 建立与断开连接

假设 a 为客户端（发送）b 为服务端（接收），下面是交互过程的简单描述。

SYN 和 FIN 只在开始和结束时使用。

3.2.1 建立连接

假设初始 X=0,Y=0，这里的序列号从 0 开始，也可使用随机数取模产生随机序列号。

```
a->b:SYN=1  ACK=0  SEQ=X=0  ACKnum=0
b->a:SYN=1  ACK=1  SEQ=Y=0  ACKnum=X+1=1
a->b:SYN=0  ACK=1  SEQ=1    ACKnum=1    第三次握手的同时发送数据
```

3.2.2 断开连接

假设初始 X=1,Y=1，模仿 TCP 四次挥手（https://www.jianshu.com/p/cd801d1b3147）但这里三次交互就够，a 不需要第二次等待，b 收到 FIN 后立即发送 FIN。

```
a->b:FIN=1  ACK=1  SEQ=X=1  ACKnum=1
b->a:FIN=1  ACK=1  SEQ=Y=1  ACKnum=2
a->b:FIN=0  ACK=1  SEQ=2   ACKnum=2
```

断开连接时，a 先发送 FIN 信号。b 接收到 FIN 之后，也发送 FIN（让对方知道自己知道了要断开连接）。a 发送 FIN, 再收到 FIN 之后, 发送 ACK（这里对方收不到可以重传），然后退出。b 发送 FIN 并且收到 ACK 之后退出。

3.3 差错检测

采用 UDP 校验和计算方法。发送端在发送分组时进行 UDP 校验和计算，结果写入 CheckSum 位。接收端接收到分组后也进行校验和计算，结果为 0xFFFF 则无错误。

计算校验和时，本实验没有添加伪首部，只是对头部和数据部分计算。首先需要把 char 型的字符数组（char 类型占 1 字节）转为 unsigned short 型的数组（unsigned short 类型占 2 字节），采用反码求和方法（加法的溢出位需要回卷），需要注意头部占 10 字节，数据部分的字节数若为奇数，需要对最后一个字节进行单独处理（可以直接拷贝到 unnsiged short 类型的数组，这与 C++ 在 x86 下属于小端编址有关）。

3.4 确认重传

在上次实验的停等协议中,发送端收到确认序号 ACKnum 后再次发送下一个分组,SEQ=ACKnum。接收端进行累积确认。如果发送端发送的分组丢失，接收端一直等待接收不发送 ACK，或者接收端发送的 ACK 丢失，都会导致发送端收不到 ACKnum，这时发送端就重传窗口内还未确认的所有分组。

3.5 基于滑动窗口的流量控制机制

本实验采用 GBN 方法，大小窗口固定为 20，注意 GBN 方法的窗口大小需要小于序列号数量即 $GBN <= 2^n - 1$，SR 方法窗口大小需要满足 $SR <= 2^{n-1}$。

滑动窗口的思想可以简单理解为，发送端一次发送多个分组。接收端依旧使用累积确认，按序接收，当收到失序的分组时直接丢弃，并发送原来的 ACKnum（期待按序收到的分组序号）。发送

端收到 ACK 后，滑动窗口。若发生超时事件或收到重复的与之前相同的 ACK，代表接收端没有收到具有 ACKnum 序号的分组，需要进行重传。

3.5.1 发送端

发送端需要响应三种类型的事件：

1. 上层调用。本实验使用的是同步机制，仅当窗口不满时才读取文件进行打包传输。

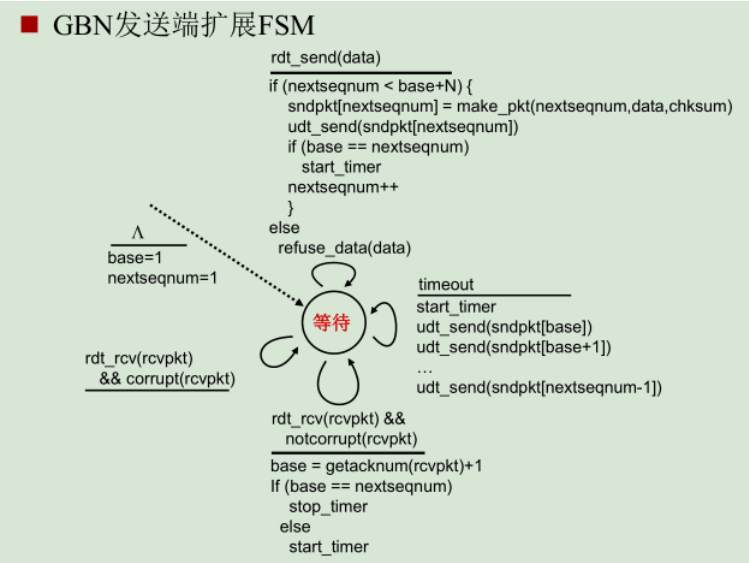
2. 收到一个 ACK（ACKnum 为 n）。代表接收端已经接收分组 n 以及之前的分组，这就是累积确认，这一点很重要，方便之后发送端窗口的加速滑动。

3. 超时事件。本实验使用的是非阻塞方式，通过设置发送和接收的超时时间，也能很好实现“定时器”的功能，并且似乎能产生加速的效果。按照 GBN 的思想，例如发送端发送多个分组，序号从 1 到 10，分组 10 发送最晚，接收端最晚接收，假设只有分组 10 损坏，只有等接收端发送完毕 ACK9 之后发送端才能知道第 10 个分组损坏，需要重传。但在编程中发现，在本实验中不使用定时器（一个原因包括可能需要涉及线程较为复杂）的非阻塞方式中，发送端可能在发送完这 10 个分组后，先收到分组 9 的 ACK 而不是分组 1 的 ACK（注意因为累积确认的原则，接收端肯定发送了 ACK2 到 ACK10），但发送端先收到 ACK10，这窗口就可以加速滑动，之前的 ACK 就可以忽略了。

发送端要负责维护一个发送分组的缓存便于之后可能的重传。

可以使用队列，队列中的元素代表着当前窗口——已发送但还未确认的分组。发送的分组都加入队列，发送后经确认的分组可以从队列移出。重传时只需重传队列中的所有分组。

下图是发送端的扩展状态机：



3.5.2 接收端

接收端的处理很简单。如果收到按序的分组 n（即上次收到的分组序号是 n-1），则接收并回复 ACKnum=n+1，接收到失序的分组，直接丢弃，并发送原来的 ACKnum（期待按序接收的分组序号）。

3.6 拥塞控制（RENO 算法）

拥塞控制是作用于网络的，它是防止过多的数据注入到网络中，避免出现网络负载过大的情况。本实验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。

设置三种状态：慢启动、拥塞避免、快速恢复。根据不同状态之间的转移来调整窗口大小。实验中做了一些调整，与 TCP 的拥塞控制有一些不同。

3.6.1 慢启动

连接建立后，进入慢启动阶段，初始拥塞窗口 CWND 大小为 1, 每当接收到一个 ACK,CWND 加 1，因此每个 RTT，CWND 翻倍。但本实验中发送端一次发送多个分组，可能只收到最后一个分组，例如发送分组 1 到 5，可能先收到分组 5 的确认，这样窗口能够加速滑动（在实验 3-2 中已经说明）。所以如果每次收到 ACK 只进行 CWND 加 1 的操作，不能正确达到“慢启动”的效果，窗口增加的过慢。

因此设置为每次收到一个 ACK，CWND 就翻倍。

3.6.2 拥塞避免

设置阈值 SSTHRESH，拥塞窗口达到阈值，就进入拥塞避免阶段。这里也是和 TCP 有一些差别。

TCP 中是对于每个 RTT，CWND 增加 1：$CWND = CWND + MSS * MSS/CWND$，也就是对于每个 ACK，CWND 只增加 $MSS/CWND$, 这是因为 TCP 是基于字节计数。

由于本实验直接使用窗口计数，且在实验 3-2 滑动窗口的实现中存在收到超前 ACK 的情况，这里设置为对于每个 ACK，CWND 增加 1。

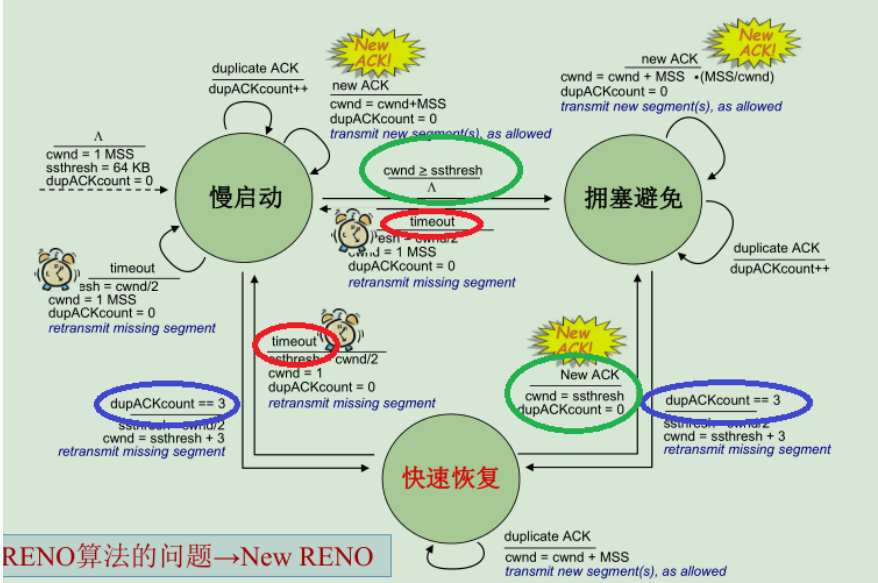
所以在慢启动和拥塞避免阶段，对于收到一个 ACK 的结果，CWND 的增长模式分别为翻倍和加 1。所以本实验与 TCP 的区别是 CWND 的变化周期是以收到一个 ACK 为单位而不是以一个 RTT 为单位。

3.6.3 快速恢复

通过三次冗余 ACK 来检测分组丢失，执行快速重传（这也已经在 3-2 实现）, 之后进入快速恢复状态。

3.6.4 三种状态的转换

下图是 reno 算法的状态机。



每种状态都有一个触发条件：

1. 无论何种状态，当发生超时事件，就进入慢启动阶段。

2. 当慢启动状态的拥塞控制窗口达到阈值，或者快速恢复状态下收到新的 ACK，就进入拥塞避免状态。

3. 无论何种状态，当收到三次冗余的 ACK，就进入快速恢复状态。

3.7 其他设计

3.7.1 定时器的设计

采用非阻塞模式，设置一个超时时间（设置为 50ms），超过此时间无响应就返回一个值（-1），这样编程更方便，也实现了“定时器”的功能。经过测试，超时未响应 recvfrom 函数的返回值为-1，错误类型 WSAGetLastError() 的返回值为 10060。

```
// 设置发送超时
setsockopt(socket_client,SOL_SOCKET,SO_SNDTIMEO, (char *)&nNetTimeout,sizeof(
    int));
//设置接收超时
setsockopt(socket_client,SOL_SOCKET,SO_RCVTIMEO, (char *)&nNetTimeout,sizeof(
    int));
```

3.7.2 随机丢包和延时的设计

使用随机数取模的方法，随机产生丢包，包括发送端丢弃数据包、丢弃重传的数据包，接收端丢弃 ACK 等等。使用 Sleep 函数进行延时。

3.7.3 快速重传和窗口加速滑动

实现了快速重传，发送端维护一个变量，代表期望接收的 ACK 序号，如果发送端收到了三次相同的冗余 ACK，就执行快速重传，立即重传当前滑动窗口内的所有分组。如果收到了超前的 ACKnum，比如发送了分组 1、2、3，期待收到 ACKnum=2、3、4，如果先收到的是 4,，就可以加速滑动窗口，而不是每次滑动一位。这种情况在实际运行中是会发生的。

3.7.4 测试方法

将输入输出重定向到文件，实现运行日志的保存。在 windows 环境下使用批处理文件实现快捷测试。

```
批处理命令:
test_server < input_server >> output_server & exit
test_client < input_client >> output_client & exit
python文件作用是将输入参数（窗口大小、文件序号）写入文件并运行批处理命令。
运行以下命令或将此命令写入bat文件运行，即可传输文件1:
python s.py 1
```

4 附录 2：实验 3-2、3-3 的代码分析

4.1 公共头文件定义

```
//设置IP和端口号等
#define server_Port 1001
#define server_IP "127.0.0.1"
#define client_Port 1002
#define client_IP "127.0.0.1"
//rand()%RAND_MOD_NUM 对随机数取模来决定发送还是丢弃分组
int RAND_MOD_NUM=15;
//非阻塞模式的超时时间
int nNetTimeout=50;//毫秒
//缓冲区大小
#define BUF_LEN 65310//比2**16小一点
#define HEADER_LEN 10//头部长度
#define DATA_LEN 65300
typedef unsigned short ushort;
/*
头部的设计
Header:10 bytes
| 2 bytes  | 2 bytes  | 2 bytes  | 2 bytes  |  2 bytes  |
|   SEQ    |  ACKnum  | CheckSum |    ID  |   Length  |
```

```
ID:2 bytes
|   FIN    |  ACK  |  SYN  |
*/
struct PACKAGE
{
    ushort SEQ;//序列号 2bytes,16bits 0-65535
    ushort ACKnum;//确认序号
    ushort CheckSum;//检验和
    ushort ID;//多个标志位
    ushort Length;//数据长度, 这是基本固定的
    char data[DATA_LEN];
};
//各个标志位
#define SYN 0x1//建立连接
#define ACK 0x2//确认
#define FIN 0x4//断开连接
```

4.2 基本函数（不传入参数，直接执行完成相应功能）

4.2.1 makePackage

用于制作分组的头部，发送端需要设置校验和、发送序列号、数据长度。接收端需要设置确认序号。

```
sendbuf.Length = length;
sendbuf.SEQ =(sendbuf.SEQ+1)%65536;
sendbuf.CheckSum = 0;
memcpy(&sendbuf, &header, HEADER_LEN);
```

注意建立连接时填充 SYN 字段，结束连接时填充 FIN 字段。发送端在读取数据后，需要再次填充 CheckSum 字段。

4.2.2 doCheckSum

计算 UDP 校验和的函数。发送端计算校验和填入 CheckSum 字段。接收端计算校验和判断是否等于 0xFFFF，不等于 0xFFFF 则数据出错。

```
//先将char型发送缓冲区转换成为16位的unsigned short型数据
//需要分情况, 考虑缓冲区长度
array = new unsigned short[count];
for (int i = 0; i < count; i++)
{
    memcpy(&array[i], &sendbuf[i * 2], 2);
}
```

```
//计算校验和
while (count--)
{
    sum += array[i++];
    sum = (sum >> 16) + (sum & 0xFFFF);
}
sendbuf.CheckSum=~sum;
```

4.2.3 Check_expected_ACK(发送端使用)

发送端检查接收的 ACKnum 是否是期望且按序的（即等于最早未确认的分组序号加 1）。如果等于期望的 ACKnum，窗口滑动一位。如果小于期望的 ACKnum，直接忽略即可。如果大于期望的 ACKnum，则窗口可以加速滑动。每次收到一个按序的 ACKnum 后，expected_ack 进行更新：$expected\_ack = recvbuf.ACKnum + 1$;。

4.2.4 CheckSEQ_EXPECTED(接收端使用)

接收端检查接收到的分组是否按序，比较接收到的分组序号是否等于刚刚发送的 ACKnum 即可。

4.2.5 IsACK,IsFIN,IsSYN

检查这些标志位是否置位。

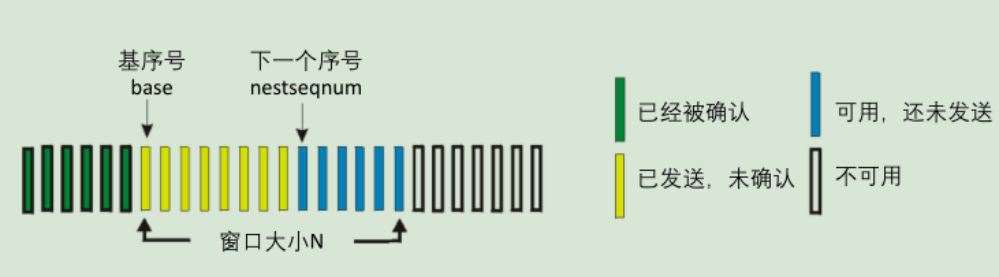
```
bool IsFIN()
{
    if ((recvbuf.ID & FIN) != 0)
        return true;
    else
        return false;
}
```

4.3 滑动窗口的发送端代码

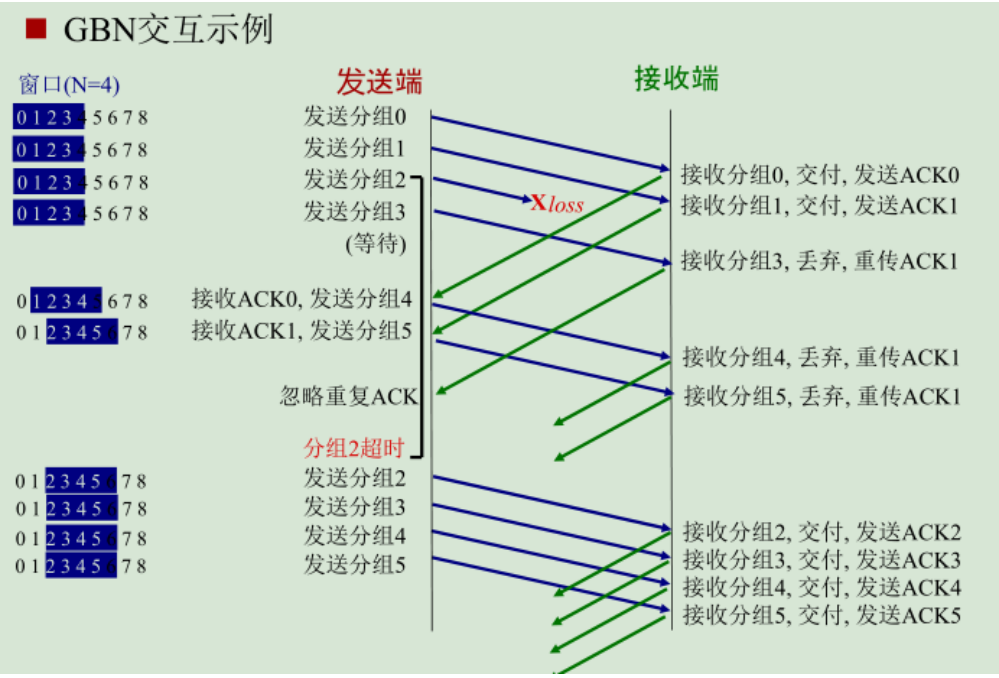
先是 WinSocket 的初始化，创建 socket，设置超时时间等操作。

第一步发送包含 SYN 的分组，准备建立连接。连接建立后，先发送文件名。之后开始读文件（每次读取数据缓冲区大小的数据量）。制作分组（包括序列号每次取模递增，数据长度，待装入数据后计算填充校验和）。发送分组，可能会随机丢弃。

发送端维护一个队列 Send_Buf_Queue 用于缓存发送的分组。滑动窗口具有 p_Base、p_NextSeqNum 两个指针，如下图所示，分别指向最早的发送未确认分组和即将发送的分组。在发送分组时，将分组加入发送缓冲队列，在接收到 ACK 后，将队头元素（已确认的分组）出队列。



GBN 的交互主要包括窗口滑动、接收 ACk、超时重传三种情况。



4.3.1 窗口滑动

初始时，base 和 nextseqnum 都为 1，先发送多个分组。收到期待的 ACKnum 后，base 加 1，收到超前的 ACKnum 后，base 增加值大于 1。base 变化后，p_NextSeqNum<p_Base+WINDOW_WIDTH 条件就满足了，进入 while 循环，窗口实现滑动。

```
/*****************读取、发送*****************
* 0 1 2 3 4 5 6 7 8 9
*   |         |    n=5(1 2 3 4 5)
*   base      next
* */
while((p_NextSeqNum<p_Base+WINDOW_WIDTH)&&read_flag)//这里是while！！！不是if
    ！！！！
{
    if(p_NextSeqNum==SEQ_SIZE)
    {//实现循环，这样也可以很简单
```

```
            p_Base-=p_Base;p_NextSeqNum-=p_Base;
        }
        file.read((char*)&sendbuf.data, DATA_LEN);
        if (file.gcount() < DATA_LEN)
        {
            //读取结束了
            sendbuf.ID |= FIN;//或操作使得FIN位置为1
            read_flag=0;//不再读取了
        }
        makePackage(file.gcount());
        //设置校验和并填充
        doCheckSum(file.gcount());
        //加入发送缓冲队列
        PACKAGE t1=sendbuf;
        Send_Buf_Queue.push(t1);
        //发送数据包,随机丢弃
        if(rand()%RAND_MOD_NUM!=0)//即遇到rand()%RAND_MOD_NUM==0丢弃
        {
            //宏替换语句, 包括发送和延时sendto(socket_client, (char*)&sendbuf, BUF
                _LEN, 0, (SOCKADDR*)&server_addr, sizeof(SOCKADDR));Sleep(SEND_TIME
                _DELAY)
            CLIENT_SEND;
            cout<<"send seq: "<<sendbuf.SEQ<<endl;//cout<<"发送第 "<<sendbuf.SEQ
                <<" 个数据包!!!"<<endl;
        }
        else
        {
            cout<<"*********随机丢弃数据包 "<<sendbuf.SEQ<<"*********"<<endl;
        }

        p_NextSeqNum++;

    }
```

4.3.2 接收 ACK

这里根据收到的 ACKnum 和期待的 ACKnum 不同，又分为三种情况。

1.recvbuf.ACKnum==expected_ack

直接移动 base，发送端缓存队列队头元素出队。

```
cout << "successfully trans " << recvbuf.ACKnum-1 << endl;
trans_bytes+=recvbuf.Length;
p_Base=recvbuf.ACKnum;
Send_Buf_Queue.pop();
```

2.recvbuf.ACKnum<expected_ack

记录一个当前收到的 ACKnum，当连续三次收到冗余的 ACK 时，就执行快速重传。

```
if(recvbuf.ACKnum==cur_ack)
{
    dup_ack_num++;
    cout << "收到之前的ACKnum且重复!!!: " << recvbuf.ACKnum << ""<< endl;
}
else
{
    dup_ack_num=0;
    cout << "收到之前的ACKnum: " << recvbuf.ACKnum << "! 不执行动作即可"<<
        endl;
    cur_ack=recvbuf.ACKnum;
}
if(dup_ack_num==2)
{
    //快速重传
    dup_ack_num=0;
    cout<<"********************执行快速重传!!! ********************"<<endl;
    goto LABEL_TRANS_AGAIN;
}
```

3.recvbuf.ACKnum>expected_ack

收到超前的 ACKnum，base 多移动几位，窗口能加速滑动。

```
cout<<"收到超前的ACKnum,可以加速滑动!!!  ACKnum: "<<recvbuf.ACKnum<<endl;
p_Base=recvbuf.ACKnum;
//p_Base移动
for (int i = 0; i < recvbuf.ACKnum-expected_ack+1; i++)
{
    Send_Buf_Queue.pop();
}
expected_ack=recvbuf.ACKnum+1;
trans_bytes+=(recvbuf.ACKnum-expected_ack+1)*DATA_LEN;
```

4.3.3 超时重传

超时未收到 ACK，重传窗口内的所有分组，只需要把发送缓冲队列里的分组依次出队列、发送、再加入队列。

```
/* base--next-1
* 0 1 2 3 4 5 6 7 8
*     |       | 已发送1 2 3 4 5
```

```
* 收到ack2，收不到ack3，可能情况：1.对方未收到seq3，期待收到seq3 2.对方收到了
    seq3，但是ack3丢失
* 都执行重传，重传2 3 4 5 6
*/
for (int i = p_Base; i < p_NextSeqNum; i++)
{
    sendbuf=Send_Buf_Queue.front();
    if(rand()%RAND_MOD_NUM!=0)
    {
        CLIENT_SEND;
    }
    else
    {
        cout<<"**********随机丢弃重传的数据包！！！！！
            **********"<<endl;
    }
    PACKAGE t1=Send_Buf_Queue.front();
    Send_Buf_Queue.push(t1);
    Send_Buf_Queue.pop();
}
cout << "超时未接收到ACK，重传 "<<p_NextSeqNum<<"-"<<p_Base<<" == "<<p_
    NextSeqNum-p_Base<<"个数据包！！！" << endl;
```

4.4 滑动窗口的接收端代码

和发送端相同，建立连接后接收文件名。之后开始接收分组，写入文件，发送 ACK 确认分组（也进行随机丢弃）。

CheckSum 函数进行校验和计算，CheckSEQ_EXPECTED 确保接收到的是按序期望的不重复的分组，否则丢弃即可，IsFIN 判断是否传输完成。

4.4.1 接收按序的分组并写入文件

```
cout << "successfully recv " << recvbuf.SEQ << endl;
trans_bytes += recvbuf.Length;
file.write(recvbuf.data, recvbuf.Length);
makePackage();
if(rand()%RAND_MOD_NUM!=0)
{
    SERVER_SEND;
}
else
{
```

```
        cout<<"**********随机丢弃ACK包！！！！！
            **********"<<endl;
    }
```

4.4.2 接收到失序的分组, 直接丢弃

```
    /**********收到失序(超前)的包
        **********
    * 比如收到了seq5, 回复acknum==6, 期望的是seq6
    * 结果收到seq8, 直接丢弃
    * package.ACKnum不变
    */
    cout<<"接收到失序的数据包 "<<recvbuf.SEQ<<" ,丢弃, 发送原ACKnum: "<<sendbuf.
        ACKnum<<"!!!"<<endl;
    if(rand()%RAND_MOD_NUM!=0)//即遇到rand()%RAND_MOD_NUM==0丢弃
    {
        SERVER_SEND;
    }
    else
    {
        cout<<"**********随机丢弃ACK包（收到失序数据包时的ACK
            ）！！！！！ **********"<<endl;
    }
```

4.5 拥塞控制代码——在发送端进行三种状态的转换

实际编程中只需要在 3-2 的基础上增加三种状态的转换即可。

定义三种状态, 并在开始时初始化为慢启动状态。

```
#define SLOW 1
#define AVOID 2
#define QUICK 3
```

慢启动状态。

```
if (recv_Ret < 0)
{
    state=SLOW;
    SSTHRESH/=2;
    CWND=1;
    cout<<"进入慢启动 CWND: "<<CWND<<endl;
    ...
}
```

拥塞避免状态。

```
if(CWND>=SSTHRESH)
{
   state=AVOID;
   cout<<"进入拥塞避免 CWND:"<<CWND<<endl;
}
```

快速恢复状态。

```
if(dup_ack_num==2)
{
    dup_ack_num=0;
    state=QUICK;
    SSTHRESH=CWND/2;
    CWND=SSTHRESH+3;
    cout<<"进入快速恢复 CWND: "<<CWND<<endl;

    goto LABEL_TRANS_AGAIN;//进行重传

}
```

收到 ACK 的动作。

```
if(state==SLOW)
{
   CWND*=2;
}
else if(state==AVOID)
{
   CWND+=1;
}
else
{
    state=AVOID;
    CWND=SSTHRESH;
    cout<<"进入拥塞避免 CWND: "<<CWND<<endl;

}
```

4.6 断开连接的其他辅助处理

4.6.1 发送端可能收不到接收端回复的 FIN

发送端在发送最后一个分组的同时发送 FIN，但有可能收不到接收端的 FIN，这时 base 和 nextseqnum 已经相等，可以根据此来决定终止连接。

```
if((p_Base==p_NextSeqNum&&p_Base!=1))
```

```
{
    cout<<"已经结束了！！！"<<endl;
    goto LABEL_Complete;
}
```

另外也可能遇到接收端退出过快，发送端甚至收不到 ACKnum。注意接收端只有接收完毕并且收到 FIN 和 ACK 才会退出。可以判断发送端重传次数，如果多次重传相同的分组，则有可能是接收端已经退出了，发送端退出即可。

```
if(p_Base==T_a&&p_NextSeqNum==T_b)
{
    dup_trans_num++;
}
else
{
    dup_trans_num=0;
    T_a=p_Base;T_b=p_NextSeqNum;
}
if(dup_trans_num==20)
{
    cout<<"对方无应答，应该是退出了，结束！！"<<endl;
    goto LABEL_Complete;
}
```

对于接收端，必须收到发送端的 FIN 和 ACK 才能退出。

```
/*发送端发来了FIN,接收端再发一次FIN，等到发送端知道了，发送ACK（发送端只有这时
    候发送ACK），之后再退出*/
while(recvbuf.ID&ACK==0)
{
    cout<<"send FIN!"<<" ";
    SERVER_SEND;//含有FIN
    do
    {
        recv_Ret = recvfrom(socket_server, (char*)&recvbuf, BUF_LEN, 0, (
            SOCKADDR *)&client_addr, &recv_para_len);
    } while (recv_Ret < 0);
}
```

5 附录 3：实验 3-2、3-3 的结果展示

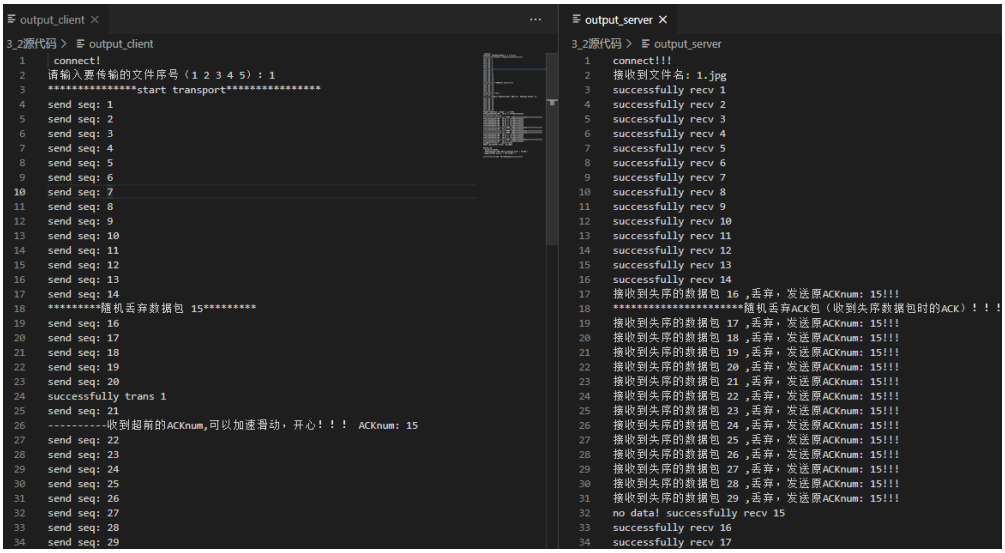
5.1 实验 3-2 结果展示

程序可以连续发送任意类型的文件，也可以选择输入窗口大小、文件名等等，能够实现文件的无损正确传输。超时时间越短（这里是 10ms），缓冲区越大（这里是比 2 的 16 次方小一些，不能超

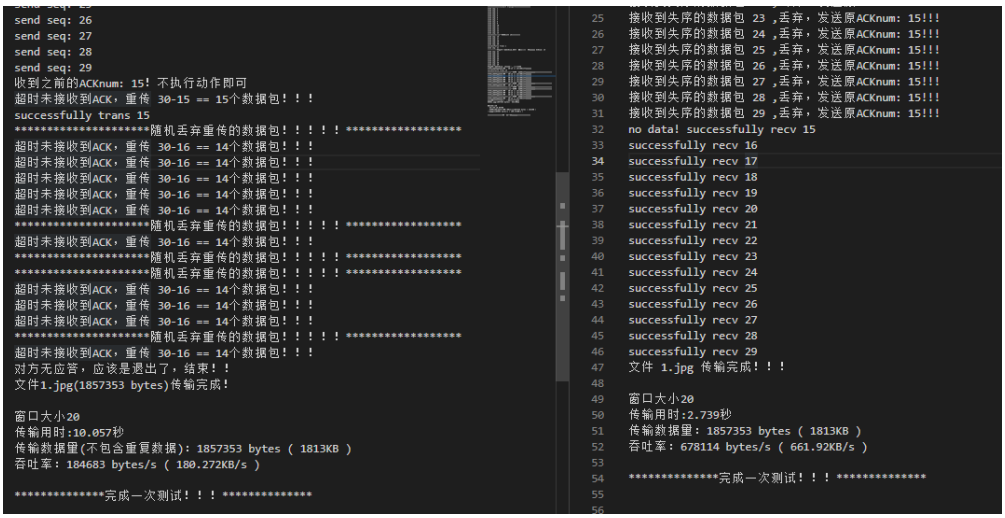
过，取 65310），文件发送速度越快。

以下是窗口大小为 20 的测试结果。

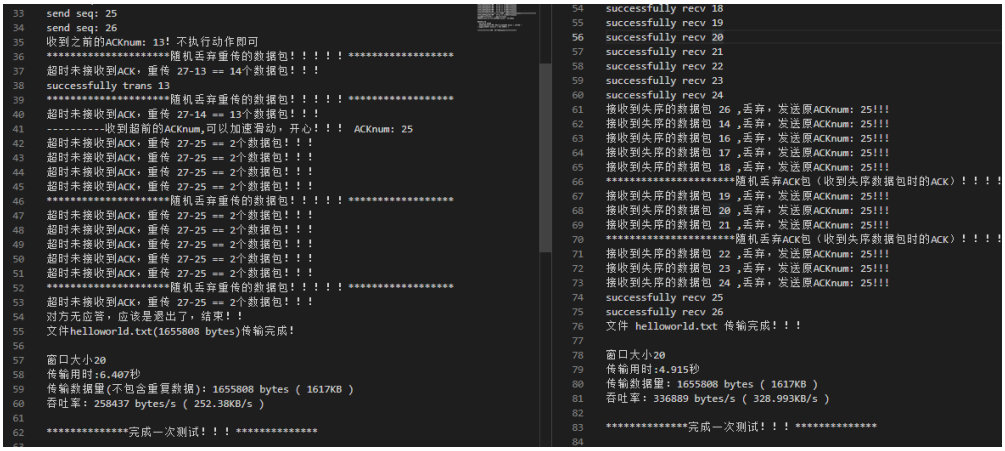
从下图可以看出，发生过发送端丢弃分组和接收端丢弃 ACK 的情况，也发生了窗口加速滑动。



下图是超时重传，可以看到第一次重传从分组 15 开始，之后分组 15 得到确认。但发送端随即丢弃了分组 16，导致第二次重传。这时也发生了接收端退出过早的情况，可以通过检查发送端是否多次重传相同分组来确认。



下图也包含了超时重传，发送端的三次重传分别从分组 13、14、25 开始。



5.2 实验 3-3 结果展示

连接建立后，拥塞控制窗口从 1 开始，先进入慢启动阶段，每收到一个 ACK，CWND 翻倍，之后 CWND 达到阈值，进入拥塞避免阶段，当收到三次冗余 ACK 还会进行快速恢复。

下面是程序运行日志。

```
connect!
请输入要传输的文件序号（1 2 3 4 5）：4
***************start transport***************
~~~~~~~~~~~~~CWND:1~~~~~~~~~~~~~~~
send seq: 1
successfully trans 1   CWND+=1 !!!
~~~~~~~~~~~~~CWND:2~~~~~~~~~~~~~~~
send seq: 2
send seq: 3
successfully trans 2   CWND+=1 !!!
~~~~~~~~~~~~~CWND:4~~~~~~~~~~~~~~~
send seq: 4
send seq: 5
send seq: 6
----------收到超前的ACKnum,可以加速滑动！！！  ACKnum: 5 CWND+=1!!!
~~~~~~~~~~~~~CWND:8~~~~~~~~~~~~~~~
send seq: 7
send seq: 8
send seq: 9
send seq: 10
send seq: 11
*********随机丢弃数据包 12*********
----------收到超前的ACKnum,可以加速滑动！！！  ACKnum: 8 CWND+=1!!!
~~~~~~~~~~~~~CWND:16~~~~~~~~~~~~~~~
######进入拥塞避免 CWND:16 ######
```

```
send seq: 13
send seq: 14
*********随机丢弃数据包 15*********
send seq: 16
send seq: 17
*********随机丢弃数据包 18*********
send seq: 19
*********随机丢弃数据包 20*********
send seq: 21
send seq: 22
send seq: 23
----------收到超前的ACKnum,可以加速滑动！！！  ACKnum: 12 CWND+=1!!!
~~~~~~~~~~~~CWND:17~~~~~~~~~~~~~~~
######进入拥塞避免 CWND:17 ######
send seq: 24
send seq: 25
send seq: 26
收到之前的ACKnum: 12！不执行动作即可
~~~~~~~~~~~~CWND:17~~~~~~~~~~~~~~~
######进入拥塞避免 CWND:17 ######
######进入慢启动 CWND: 17 ######
超时未接收到ACK, 重传 27-12 == 15个数据包！！！
----------收到超前的ACKnum,可以加速滑动！！！  ACKnum: 14 CWND+=1!!!
~~~~~~~~~~~~CWND:2~~~~~~~~~~~~~~~~
######进入慢启动 CWND: 2 ######
超时未接收到ACK, 重传 27-14 == 13个数据包！！！
######进入慢启动 CWND: 1 ######
超时未接收到ACK, 重传 27-14 == 13个数据包！！！
######进入慢启动 CWND: 1 ######
超时未接收到ACK, 重传 27-14 == 13个数据包！！！
######进入慢启动 CWND: 1 ######
文件helloworld.txt(1655808 bytes)传输完成！

传输用时:5.275秒
传输数据量(不包含重复数据): 1655808 bytes ( 1617KB )
吞吐率: 313897 bytes/s ( 306.54KB/s )

**************完成一次测试！！！**************
```