

# UDP 可靠传输 Part4

2014074 费泽锟

年级：2020 级

专业：信息安全

指导教师：徐敬东、张建忠

## 摘要

在 UDP 实现可靠传输 Part4 的部分之中，这一部分之中设计与实现不再是重点，对于 UDP 可靠传输实现流程之中的过程的对比分析为本部分的重点内容，从停等协议的实现，到基于滑动窗口思想的 GBN 算法与 SR 算法的实现，最终为基于拥塞控制的算法实现。

在分析之中，会主要以丢包率以及延迟时间作为变量，以文件传输的吞吐率和时延作为性能测试指标，除了对不同机制的性能对比分析之外，本部分还会针对一些因为本机传输本机的特殊实验环境出现的现象进一步进行分析。

由于 router 程序在 Windows 11 操作系统中总是出现些许问题，所以这里的丢包率和模拟 router 转发的子线程均为自己编写实现的。

**关键词：**UDP 可靠传输；丢包率；延迟时间；吞吐率；时延；

## 一、从停等机制到滑动窗口

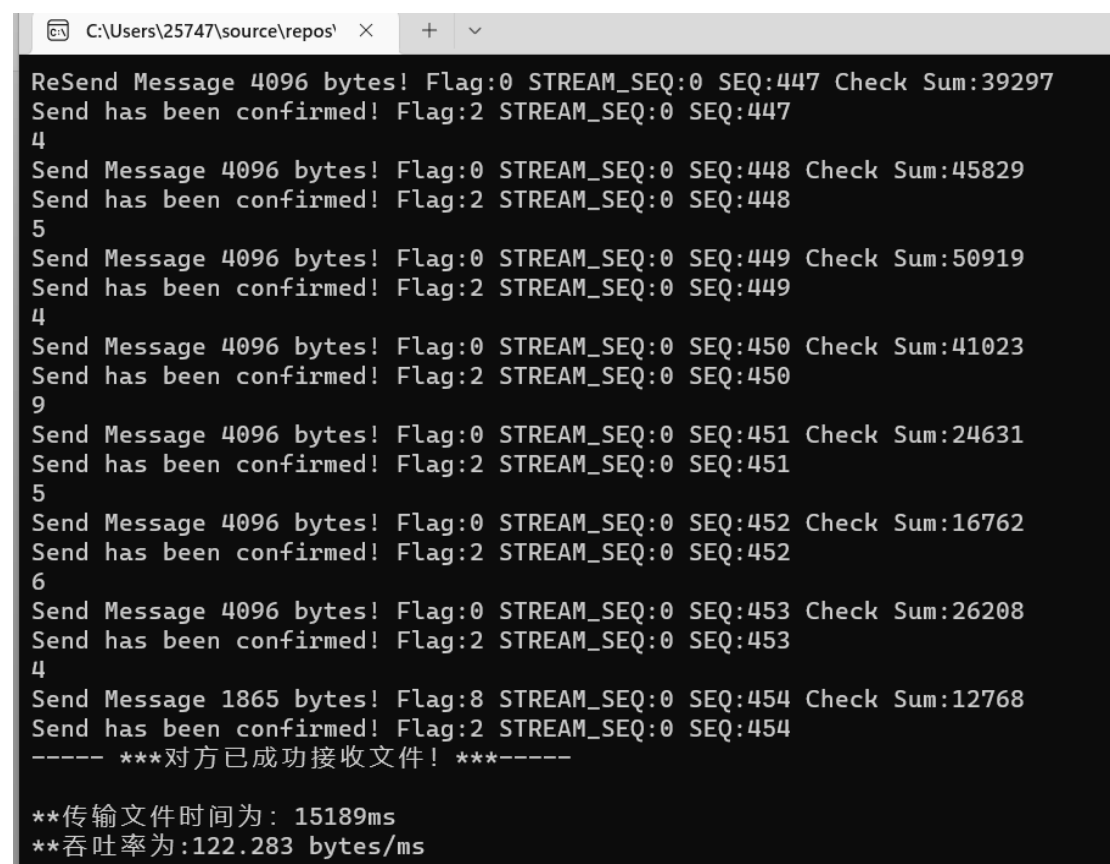
在这一部分的设计之中，主要体现的就是从停等机制到有点类似于流水线的滑动窗口机制的转变，对于停等机制，在每次发送一个数据分组之后都会一直等待接收端返回相应的 ACK 响应才会继续发送下一个数据分组，而滑动窗口则是

会将窗口内部的所有数据分组都进行发送，当窗口的 base 也就是按顺序下一个发送等待接收的数据分组，GBN 并不带有缓冲区所以窗口内的所有数据分组都默认丢失，而 SR 则带有缓冲区。GBN 只有一个计时器所以当超时时将窗口内的所有数据分组都进行重传，而 SR 则是对窗口内部每一个数据都设置一个计时器，如果对应计时器超时那么重传对应分组即可，因为每个数据分组按道理来说都应当设置一个新线程有点复杂，所以这里的实验分析主要基于的是完整的 GBN 算法实现。

首先在这一部分之中，并没有涉及到滑动窗口中不同的窗口大小的问题，所以这里使用默认的窗口大小，也就是 10 进行实验，为了避免实验的偶然性，这里会对每一个实验进行五次的时延和吞吐率的测试，取其平均值为最终结果，数据分组的大小为 4096 字节，也就是一个页的大小）。

首先就是不设置任何延迟时间和丢包率进行实验测试（均使用测试文件中的 1.jpg 文件进行测试）（1.jpg 的大小为 1857353 字节）。

取其中某次成功的实验截图如下：



```
C:\Users\25747\source\repos\ × + v
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:447 Check Sum:39297
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:447
4
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:448 Check Sum:45829
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:448
5
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:449 Check Sum:50919
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:449
4
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:450 Check Sum:41023
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:450
9
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:451 Check Sum:24631
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:451
5
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:452 Check Sum:16762
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:452
6
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:453 Check Sum:26208
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:453
4
Send Message 1865 bytes! Flag:8 STREAM_SEQ:0 SEQ:454 Check Sum:12768
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:454
----- ***对方已成功接收文件! ***-----

**传输文件时间为: 15189ms
**吞吐率为:122.283 bytes/ms
```

对于每次实验进行五次重复实验，取其平均值作为最终结果，以避免偶然情况的发生，最后一行为平均值结果，以下为统计数据：

停等机制吞吐率	停等机制时延	滑动窗口吞吐率	滑动窗口时延
122.283 bytes/ms	15189ms	3310.79 bytes/ms	561ms
107.009 bytes/ms	17357ms	3571.83 bytes/ms	520ms
112.519 bytes/ms	16507ms	3511.06 bytes/ms	529ms
105.874 bytes/ms	17543ms	3544.57 bytes/ms	524ms
105.904 bytes/ms	17538ms	3264.24 bytes/ms	569ms
110.380 bytes/ms	16,826.8ms	3435.73 bytes/ms	540.6ms

这里使用 ms 作为时间单位，在此前的设计之中均使用的是 s 作为单位，但是对于对比实验而言，使用 ms 作为单位更加精细细致，所以这里使用 ms 作为时间单位。

可以看到如果在没有任何是延迟时间以及丢包率的情形下，滑动窗口机制的传输效率与停等机制相比大大提升了（窗口大小为 5，显然本机传输本机 5 的大小并不会因为数据分组过多而导致丢包）。从结果的数值停等机制为 16826.8ms，而滑动窗口则仅仅为 540.6ms，如果没有丢包率和延迟时间的影响，滑动窗口的效率几乎为停等机制的 31 倍，这里为什么窗口大小为 5 倍而效率改变为 31 倍呢，我们只需要想明白因为在本机传输本机时如果又没有延迟时间，又没有丢包率从那么最耗时间的就是 UDP 向下封装发送的过程了，停等机制发送时需要经历这样的阶段，接收时还要等待这样一个阶段，而滑动窗口则是将这一部分时间重叠了不少（不管是单线程亦或是多线程），所以能够有大幅度的效率提升。

接下来我们来看看在丢包率不断增大的情形下，两种机制的表现是怎样的，注意此时并不对延迟时间进行设置，同样的对每次实验进行五次重复实验取平均值，但是因为此时的表格有点多所以这里对于表格中的统计结果就不再进行展示了，仅仅展示不同丢包率下的传输效率的对比表格以及图形结果，展示如下：

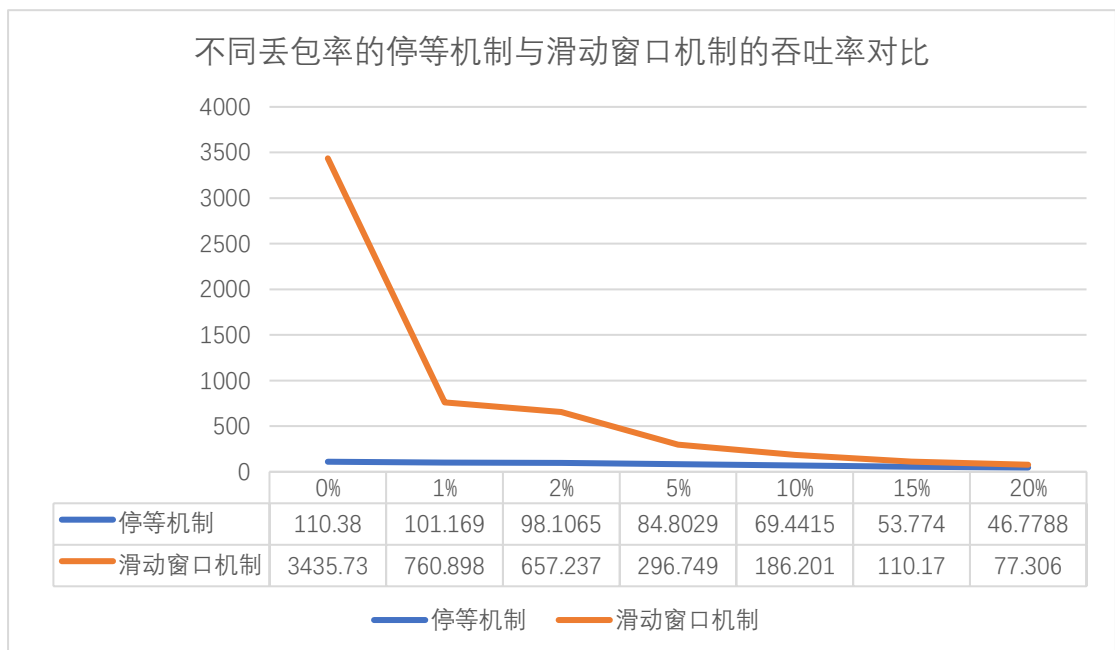
丢包率	停等机制吞吐率	停等机制时延
1%	101.169 bytes/ms	18359ms

2%	98.1065 bytes/ms	18932ms
5%	84.8029 bytes/ms	21902ms
10%	69.4415 bytes/ms	26747ms
15%	53.774 bytes/ms	34540ms
20%	46.7788 bytes/ms	39705ms

丢包率	滑动窗口吞吐率	滑动窗口时延
1%	760.898 bytes/ms	2441ms
2%	657.237 bytes/ms	2826ms
5%	296.749 bytes/ms	6259ms
10%	186.201 bytes/ms	9975ms
15%	110.17 bytes/ms	16859ms
20%	77.306 bytes/ms	24026ms

在所有实验的过程之中，设置的计时器的超时重传时间都是一样的，均设置的为 200ms，以保证没有额外因素的影响。因为现实世界之中，最大的丢包率也就在 5%左右，所以这里实验之中最大丢包率到 20%就截止了。

可以得到在延迟时间相同的情形下，仅仅改变丢包率的实验结果，将停等机制与滑动窗口进行对比可以得到结果如下图所示：



可以观察发现到，因为停等机制一直需要等待响应的 ACK 消息，所以就算

有丢包的情形需要继续等待的时间也不是很长，所以影响没有滑动窗口的影响那么大，滑动窗口收到了丢包率的极大影响，当丢包率达到 20%时基本与停等机制的效率相同，因为滑动窗口 GBN 还要重新发送窗口内的所有数据分组，可以设想如果丢包率进一步扩大很有可能出现滑动窗口的效率更低的情形，因为重传的过程之中也有可能继续丢包，这样就需要再次重传了。

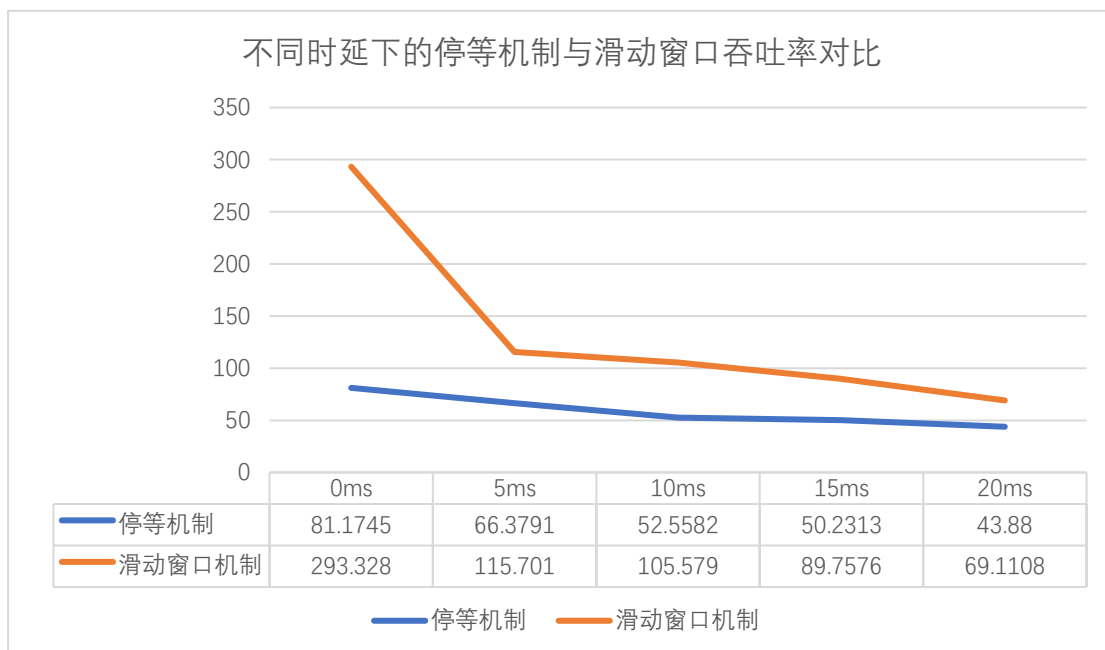
接下来我们继续进行延迟时间的实验，固定丢包率为接近现实实际的 5%的丢包率，改变延迟时间仍然采用五次重复实验，最终使用平均值作为结果，图形化分析展示则使用吞吐率这一比较热门的指标。

延迟时间	停等机制吞吐率	停等机制时延
0ms	81.1745 bytes/ms	22881ms
5ms	66.3791 bytes/ms	27981ms
10ms	52.5582 bytes/ms	35339ms
15ms	50.2313 bytes/ms	36976ms
20ms	43.88 bytes/ms	42328ms

延迟时间	滑动窗口吞吐率	滑动窗口时延
0ms	293.328 bytes/ms	6332ms
5ms	115.701 bytes/ms	16053ms
10ms	105.579 bytes/ms	17592ms
15ms	89.7576 bytes/ms	20693ms
20ms	69.1108 bytes/ms	26875ms

因为 router.exe 不能使用的原因，这里我使用的是自己在接收端设置的模拟 router 的转发子线程，如果延迟时间太大是有可能出现丢包的情形的。

可以得到在 5%的丢包率的情形下，不同的时延时停等机制与滑动窗口的对比如下：



可以看到除了 0ms 到 5ms 时传输效率有一个阶跃，其实无论是在 5ms 还是 20ms 时的滑动窗口的吞吐率差的绝对值其实是类似的，是因为大家的延迟时间是相同的，所以发送每个数据分组到达的时间其实是差不多的，但是滑动窗口可能会重传更多的数据分组而导致有更长的时延，所以这时的吞吐率的斜率较为平缓。

同样的我们能看到，从 0ms 到 5ms 有一个阶跃下降，斜率较大，是因为此时还是 UDP 向下封装传输占据了大部分的时间，而停等机制需要一直等 ACK 响应的原因，该原因与前一部分相同。

至此，停等机制与滑动窗口机制的分析基本完成，我们能够成功发现，相较于停等机制如果采取大小为 5 的窗口，10% 的丢包率能够有，5ms 延迟时间，能够有接近 3 倍的传输效率提升。

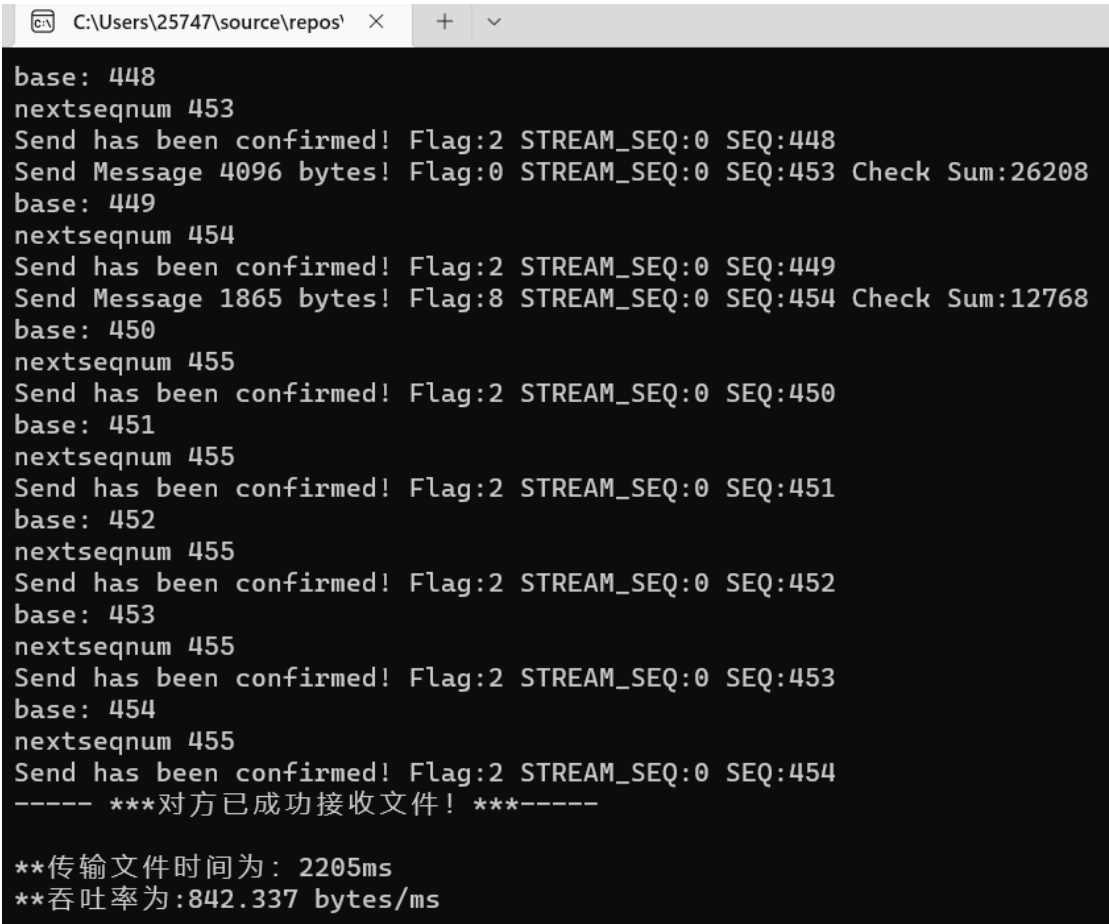
## 二、 滑动窗口机制分析实验

接下来这一部分的实验分析主要是针对在滑动窗口中的不同的滑动窗口大小来进行对比分析的。滑动窗口太小，无法充分发挥滑动窗口的优势，快速地传输文件，而滑动窗口太大又容易出现转发时的线程缓冲区填满而导致的额外的丢包现象，所以说如果单纯地使用滑动窗口机制，那么选择一个合适的滑动窗口大

小是一个值得我们讨论的话题。

接下来我将会设置不同的滑动窗口的大小来看看，不同滑动窗口的大小在不同的丢包率和不同的延迟时间下的表现是怎样的，选取的窗口大小分别为 2、5、10、20 来进行实验分析。

同样的我们先给出一个传输成功的截图：



```
base: 448
nextseqnum 453
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:448
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:453 Check Sum:26208
base: 449
nextseqnum 454
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:449
Send Message 1865 bytes! Flag:8 STREAM_SEQ:0 SEQ:454 Check Sum:12768
base: 450
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:450
base: 451
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:451
base: 452
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:452
base: 453
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:453
base: 454
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:454
----- ***对方已成功接收文件! ***-----

**传输文件时间为: 2205ms
**吞吐率为: 842.337 bytes/ms
```

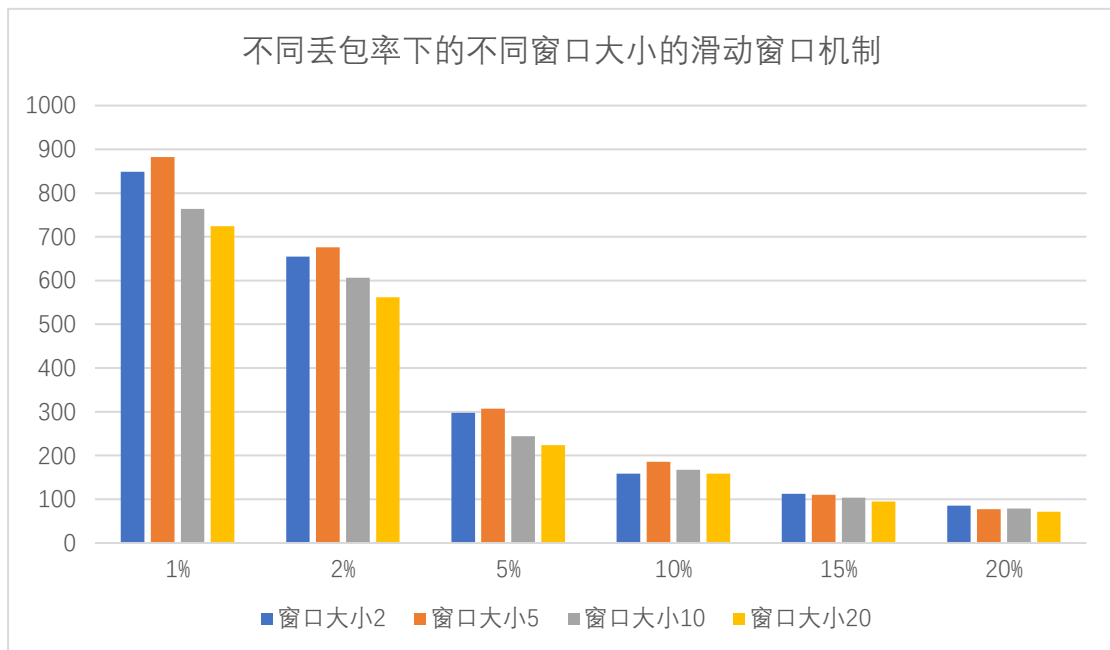
接下来给出在相同的延迟时间下，不同窗口大小下的不同丢包率的传输效率的统计数据，统计数据如下，这里就不给出时延的结果了因为时延\*吞吐率是文件大小这一个定值，仅仅给出吞吐率作为展示指标，同样采取五次实验结果的平均值为统计结果：

丢包率	2	5	10	20
0%	3148.06	3304.90	3465.21	3287.35
1%	848.882	862.337	763.714	724.102
2%	654.920	656.077	606.78	561.473
5%	297.510	327.034	243.939	223.454

10%	158.558	185.735	167.269	158.317
15%	112.274	109.948	103.856	94.7242
20%	85.7159	77.3865	79.0834	71.4174

我们可以看到其实当丢包率为 0 的时候，基本上窗口大小对于传输效率的影响是微乎其微的，因为基本上发送线程能够很快的将窗口内的数据分组全部发送完毕，这是 CPU 执行时间的数量级也就是 ns 级别的，而发送后等待接收 ACK 后才会将窗口滑动，其实当窗口发满之后就是一个一个向后发送的，那个时间段就和停等的形式很像了。

但是当出现了丢包的情形后，窗口大小就会一定程度上影响传输的效率了，为了更好地能够分析丢包率，窗口大小以及传输效率之间的关系，下图给出条形图对比（为了更好的体现对比，就不列出 0% 时的数据了）：



图中就不给出数据表了，因为数据表过大，数据表的内容就是条形图上方的数据表格，我们可以清晰地发现，在丢包率小于等于 10% 的时候，对比窗口大小为 2 和窗口大小为 5 的数据，可以发现此时增大数据窗口能够获得一定的传输效率提升，但是我们可以发现窗口大小超过 10 之后反而没有获得传输效率的提升，这是因为窗口大小变大之后，那么每次丢包需要重传的数据分组就会增加，重传的数据分组量增加就带来更多的时间消耗，并且多重传的数据分组也会带来多的丢包情形，从而导致传输效率的下降，也就是说其实不同的丢包率下的最佳的滑动窗口大小是不相同的。



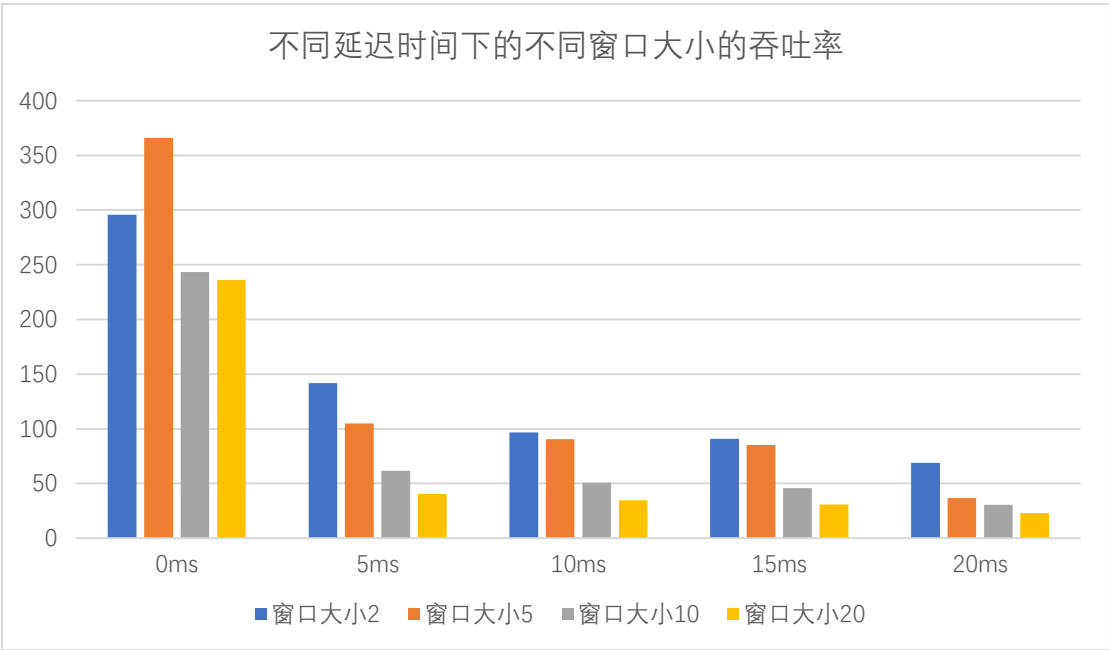
但是根据图中显示的那样，当丢包率超过 10%之后，其实反而窗口大小为 2 与窗口大小为 5 对比，是大小为 1 更有优势，这说明此时的丢包率过大了，只有很小的窗口大小才有更好的效率，也就是说数据分组的传输等待时间此时已经小于了因为丢包出现的时间消耗。

其实，我们也可以看到，当丢包率大于 10%时，不同窗口大小的传输效率已经几乎相同了，此时因为丢包而产生的时间消耗占据了主导地位。

接着我们来继续进行延迟时间的实验分析，现在我们固定丢包率为 5%不变（最贴近实际情况），接着通过改变延迟时间观察各个数据分组是否保持着相似的图线。统计数据如下：

延迟时间	2	5	10	20
0ms	295.71	365.909	243.428	235.993
5ms	141.696	104.906	61.6046	40.3285
10ms	96.7069	90.5099	50.615	34.612
15ms	90.762	85.321	45.6769	30.6624
20ms	68.9748	36.5194	30.4132	22.858

通过直接观察上述的数据还是很难分析的，我们还是先获得图形化的结果。



我们来观察统计得到的图形化结果，可以发现本来窗口大小为 5 还有不错的优势，但是当开始有转发的延迟时间时就会出现其实每次重发所需要的时间增多了，也就和丢包率部分的分析是相同的，每次重发的时间消耗增加且有可能伴随

着重发次数的增多（缓冲区已满导致数据分组的丢失），就会让丢包的时间消耗占据主导地位，从图中我们就可以看出当有延迟之后窗口大小 5 就不再占据着优势了。

同时我们还能看到一个现象，那就是对于窗口大小 2 和窗口大小 5 呈现出的是两次下降，而窗口大小 10 和窗口大小 20 呈现的是单次下降，下降的更为明显，这说明窗口大小 10 或 20 导致了在转发线程之中的缓冲区的填满而导致了更多的数据分组丢失，所以下降的会更快一点（实际上在我设计模拟 router 的转发线程时就设置了一个可以被填满的缓冲区大小）。

### 三、 从滑动窗口到拥塞控制

我们从滑动窗口的分析之中已经可以得知，其实滑动窗口的大小选取和实际的丢包率以及延迟时间是息息相关的，也就是说如果能在当时的丢包率下选择合适的窗口大小就能够达到较好的传输效率。这一思想与拥塞控制的思想不谋而合，对于拥塞控制而言，就是一个逐步试探、探测的过程，如果网络实际条件好就会逐步增大拥塞窗口的大小，滑动窗口由拥塞窗口和接收端的通告窗口所决定，一般来说通告窗口都不会太小，而如果出现了丢包或者拥塞的现象，拥塞控制会通过减小拥塞窗口的大小来控制数据分组的发送情况。

所以其实拥塞控制有点类似于预测网络的情形，理论上来讲拥塞控制能够带来不错的性能提升。

还是先给出拥塞控制传输成功的截图：

```
C:\Users\25747\source\repos\ X + v
***** 拥塞避免阶段 *****
Send Window Size: 10
base: 451
nextseqnum 453
*** 拥塞避免阶段, 线性递增+1 ***
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:451
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:453 Check Sum:26208
***** 拥塞避免阶段 *****
Send Window Size: 11
base: 452
nextseqnum 454
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:452
Send Message 1865 bytes! Flag:8 STREAM_SEQ:0 SEQ:454 Check Sum:12768
***** 拥塞避免阶段 *****
Send Window Size: 11
base: 453
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:453
***** 拥塞避免阶段 *****
Send Window Size: 11
base: 454
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:454
----- ***对方已成功接收文件! ***-----

**传输文件时间为: 2210ms
**吞吐率为:840.431 bytes/ms
```

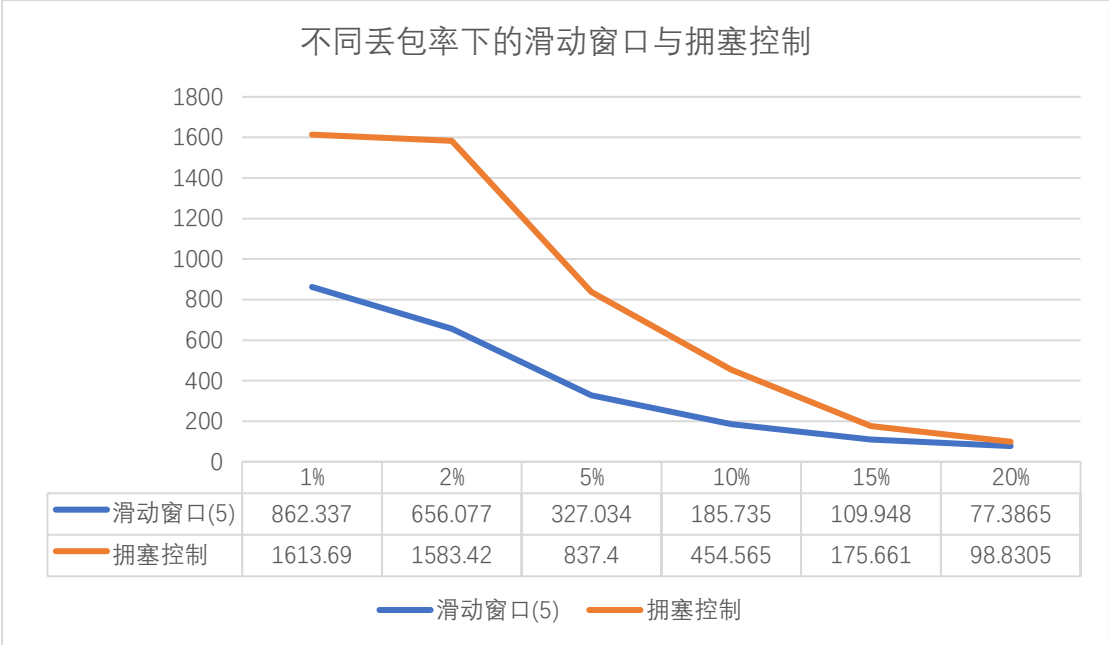
接下来确定滑动窗口机制的参数，固定窗口大小为分析二中性能表现最好的5，固定滑动窗口大小为5，而该部分的拥塞控制算法为为了适配GBN而实现的改进的New Reno算法，我们首先固定延迟时间，选择不同的丢包率来进行比较分析，对于滑动窗口的数据采取第二部分之中重复实验的数据，统计结果如下（同样以吞吐率为主要指标）：

丢包率	滑动窗口机制（5）	拥塞控制
0%	3304.90	2920.37
1%	862.337	1613.69
2%	656.077	1583.42
5%	327.034	837.4
10%	185.735	414.565
15%	109.948	175.661
20%	77.3865	98.8305

首先我们可以看到在不会丢包的情形下，因为滑动窗口始终保持着5的大小，并且拥塞控制在慢启动阶段每次接收一个新的ACK就会增加1，但是在拥塞避

免阶段是线性递增，增加的速度不快，且 1.jpg 的大小不是很大，这些因素共同作用导致滑动窗口在丢包率为 0 的情况下效率比拥塞控制高。

接下来我们来看看统计数据得到的图形化结果：



接下来我们就能够发现无论丢包率在 20% 以下是多少，传输的效率都有了不错的提升，在丢包率小于 10% 的情形下，传输效率有 2-3 倍的提升，而在 10%-20% 的丢包率时则是约有 50% 左右的效率提升。

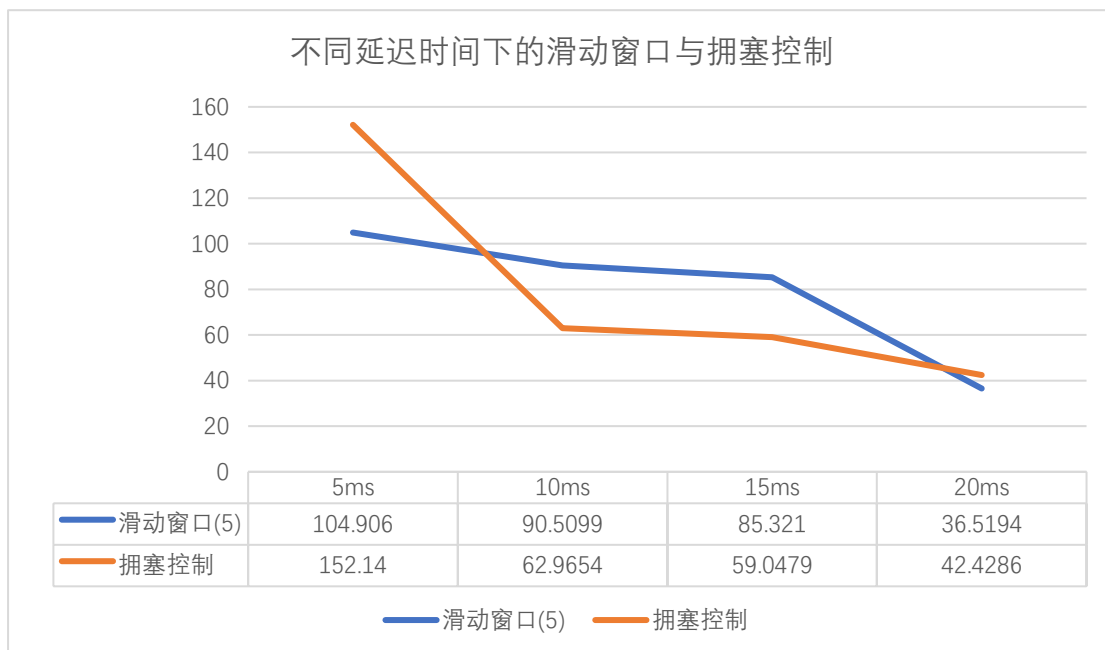
在实验过程之中通过仔细的观察，发现其实每次传输过程之中在丢包率大于 5% 以后，基本上窗口大小波动的最大值在 10 左右，10 的中位数即为 5，所以能更加体现出拥塞控制的探测作用，也就是说在没有出现随机丢包时，总是认为网络情况是好的，所以会不断地增加数据分组的发送数量，而当出现丢包的情形之后就会认为网络情况变差，就会减小窗口大小。并且效率的提升还取决于三次重复的 ACK 响应之后就会认为出现了丢包的现象，在我的改进设计之中为了优化性能，也进行了独立的设计（也就是快速恢复阶段的设计，详细请见 UDP 可靠传输 3-3），提前判断丢包现象就可以避免计时器的超时重传了。

所以，可以发现使用了改进的 New Reno 拥塞控制算法后，性能有了很不错的提升，对于 1.jpg 通常 3-4 秒就可以完成传输。

最后就是在固定丢包率之后，进行延迟时间的对比分析实验，同样的我们固定丢包率为 5% 恒定，以下为延迟时间改变的统计数据（同样的，对于滑动窗口而言采用第二部分的统计数据）（同样采取五次重复实验取平均值为统计结果）：

延迟时间	滑动窗口（5）	拥塞控制
0ms	365.909	816.348
5ms	104.906	152.14
10ms	90.5099	62.9654
15ms	85.321	59.0479
20ms	36.5194	42.4286

我们已经取得了最后一次实验的结果，下图为得到的图形化结果：



我们分析这个图表的结果，这个图表的结果是有点点奇怪的，在 5ms 延迟时间的时候还能够有 50% 左右的性能提升，但是在 10ms 的延迟时间时复杂的策略就已经变成了累赘了。

也就是说在延迟时间大于 10ms 之后，经过观察分析，拥塞控制的状态经常在快速恢复阶段和慢启动阶段左右横跳，也就是说滑动窗口一直维持在一个相对较小的水准，但是很有可能是因为我快速恢复阶段的设计，也就是会将所有的窗口内的数据分组重新发送一遍，也就是说这有可能会网络之中存在更多的数据分组，而导致更多因为重传丢包导致的时间消耗出现。

至此，所有的 UDP 可靠传输之中的不同机制的对比实验分析就已经完成了，接下来将在下一部分之中提到一些在实现过程之中遇到的问题。

## 四、 一些有趣的分析

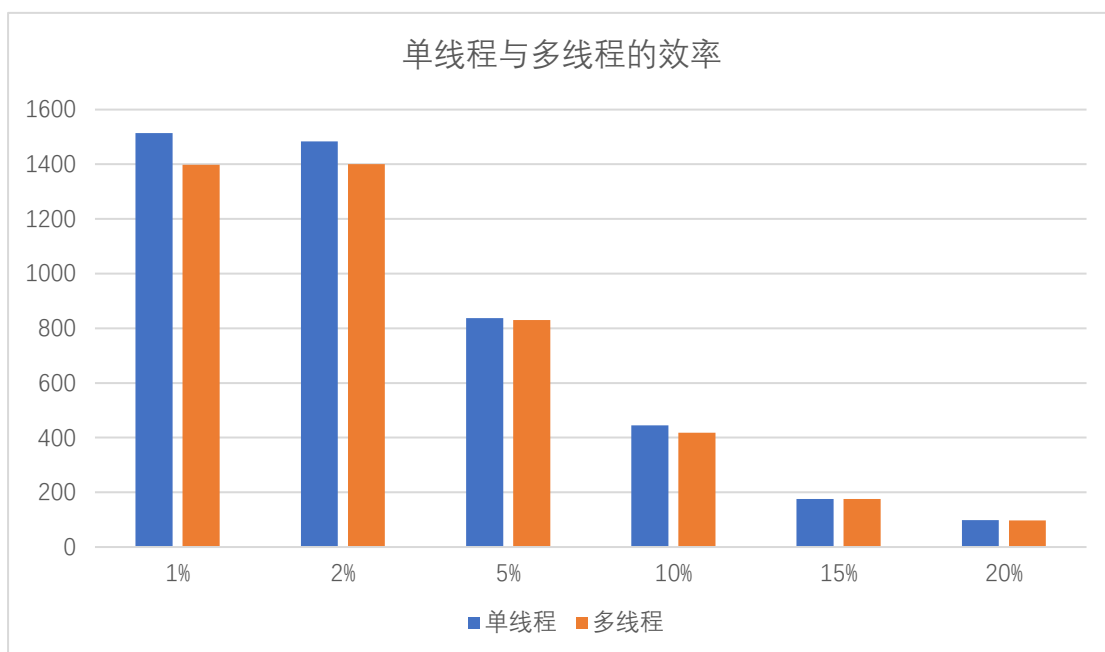
其实在这一部分之中，将会提到两部分的内容，其中一部分的内容在前面的分析之中已经分析过了，那就是如果在丢包率为 0 或者较小的时候，滑动窗口都会快速发满，因为 UDP 封装和发送过程的数量级通常为 ms 级别的，而 CPU 运行程序的数量级通常为 ns 级别的，当滑动窗口发满后就会变成发送一个数据分组确认一个数据分组的模式，详情可见滑动窗口实验的分析。

另一个想要提到的问题就是在实现单线程和双线程的效率问题，按照理论来说，多线程保持了原本的发送与接受的异步操作，而如果一个单线程则是将异步操作变成了同步操作，也就是接收缓冲区如果接收到了新的 ACK 响应消息那么会在下一个发送循环来进行处理。

按道理来说其实多线程的效率应当比单线程更好一点，但是统计的结果如下，统计的是在不同丢包率下的单线程与多线程的拥塞控制传输效率：

丢包率	单线程拥塞控制	多线程拥塞控制
0%	2900.15	2854.23
1%	1513.69	1398.65
2%	1483.42	1400.28
5%	837.4	830.62
10%	444.565	418.281
15%	175.661	176.339
20%	98.8305	97.5471

我们还是将其转换成图形化的结果看看：



可以发现其实在丢包率较小的情形下，反而是单线程更加快速，这个问题经过思考之后，应该是因为本机对本机的传输在延迟设置为 0 的时候，那么基本是没有什么真正的延迟的，毕竟本机传输本机而不是实际的应用场景，那么这个时候为了保证多线程的运行，我们不仅要加入对临界区的上锁解锁操作，我们还要考虑的是多线程可能存在的操作系统对其的上下文切换操作，这个操作向比较 CPU 的运行时间是很长的，所以很有可能是这些原因导致的在丢包率和延迟影响不大的情况下，单线程反而比多线程的传输效率还高一点。

## 五、 总结

在本次实验之中，深入地对比分析了停等机制、滑动窗口机制和拥塞控制机制的差异，并且对实验过程之中特殊的实验环境导致的问题进行了分析。在实现过程之中，对于多线程与单线程的相关知识进行了深入了解。

参考 TCP 的机制，我们仍需要对 UDP 可靠传输的设计进行一定的改编与优化，对于最后一次的实验分析就先告一段落，对于未来可能的优化工作主要是集中在设置缓冲区的问题上。

这里给出的是个人的 **Github** 仓库链接：<https://github.com/FZaKK/Computer-Networking>