南開大學
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

Lab03-04：性能对比实验

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 10 日

目录

1 实验要求

基于给定的实验测试环境，通过改变延时和丢包率，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。

- 控制变量法: 对比时要控制单一变量 (算法、窗口大小、延时、丢包率);
- Router: 可能会有较大延时, 传输速率不作为评分依据, 也可自行设计;
- 延时、丢包率对比设置: 要有梯度 (例如 30ms,50ms, …5%,10%...);
- 测试文件: 必须使用助教发的测试文件 1.jpg、2.jpg、3.jpg、helloworld.txt;
- 性能测试指标: 时延、吞吐率, 要给出**图**、**表**并进行分析

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 测试说明

首先，在实验之前，简单回顾一下我们前三个实验的设计机制。**有关本人的设计情况，在前三次实验报告中均已经详细说明，若需要查看，请查看我的 Lab03-01、Lab03-02、Lab03-03 三个实验报告**。

3.1 Lab03-01: 停等机制

停等协议是最基础的数据传输方法，核心思想就是在发送每个数据包后等待确认回复，再发送下一个数据包。这种方法虽然简单，但效率较低，尤其是在高延迟的网络环境中。

3.2 Lab03-02: 滑动窗口机制

GBN (Go-Back-N) 协议引入了滑动窗口机制，允许发送端在等待确认的同时发送多个数据包。这大大提高了数据传输的效率。然而，GBN 协议在遇到丢包时需要重传整个窗口内的所有数据包，可能导致重复传输已被接收端正确接收的数据，从而降低了网络资源的使用效率。

3.3 Lab03-03: 拥塞控制机制

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。该机制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。在丢包率或者时延较大的情况下，该方法的性能十分优越。

3.4 测试机制

由于 router 的误差过大，所以在本学期的实验中，我都自己设计了对应的延时和丢包机制，下面我来一一进行说明。

3.4.1 延时机制

我们主要通过 sleep 函数来进行实现，通过控制输入的延时变量，在程序中进行睡眠，来达到我们的延时效果。具体的代码实现如下所示：

```
// 获取用户输入的绝对延迟时间（毫秒）
void getLatencyMillSeconds() {
    std::cout << "Please input the latency of time in transfer(0-1000ms):" <<
    ↪ std::endl;
    std::cin >> Latency_mill_seconds;
    while (Latency_mill_seconds < 0 || Latency_mill_seconds > 1000) {
        std::cout << "out of range, please input again." << std::endl;
        std::cin >> Latency_mill_seconds;
    }
}
```

该部分用于获取我们预先设定的程序延时。

```
if (Latency_mill_seconds) {
    Sleep(Latency_mill_seconds);
}
```

该部分在程序中调用，如果需要延时，则调用 Sleep 函数，来进行对应的延时处理。

3.4.2 丢包机制

对于丢包机制的实现，我们使用了随机数生成的方法。

首先，我们在程序执行时，获取到了丢包率的大小输入，代码如下所示：

```
// 获取用户输入的数据包丢失率
void getPacketLossRate() {
    std::cout << "-----------The Loss of Packet-----------" << std::endl;
    std::cout << "Please input the loss of packet in transfer(0-100):" <<
    ↪ std::endl;
    std::cin >> Packet_loss_range;
}
```

然后再程序中，我们通过生成随机数进行大小比较的方法来实现了我们的随机丢包的功能，代码如下所示：

```
// 模拟丢包和延迟，如有一定概率随机触发延迟调整
int random_number = rand() % 100;
if (random_number < Packet_loss_range) {
    lock_guard<mutex> log_queue_lock(log_queue_mutex);
    log_queue.push_back("------------RELATIVE DELAY TIME!-----------\n");
    // 动态调整超时时间：延长超时时间，用于模拟网络延迟情况
    client_timer.set_timeout(Latency_param * client_timer.get_timeout());
}
```

这样，我们就完成了我们的丢包率的设计。

3.5 测试标准

我们采用控制变量法的方式，来对我们的三组机制进行性能的对比。我们一般都是改变丢包率或者时延，来达到我们的梯度设计。对于测试文件，本实验全部采用文件 2, 进行测试，对于性能的测试指标，我们考虑时延和吞吐率这两个，实际上这两个是成反比的。

4 停等机制 & 滑动窗口机制性能对比

4.1 实验设置

本组实验主要用来测试 Lab03-01 和 Lab03-02 的性能对比，我们主要通过改变其丢包率和延时来进行测试。

首先，我们统一一下本组实验的测试标准。本组实验，我们使用测试文件 2.png 进行测试，对于停等机制和滑动窗口 GBN 机制进行对比，**首先，控制延时为 0ms，测试在不同丢包率（梯度设置为 0%,20%,40%,60%,80%）下的传输性能；然后我们控制丢包率为 0，测试在不同延时（梯度设置为 0ms,120ms,240ms,360ms,480ms）下的传输性能**。

4.2 不同丢包率下的性能对比

4.2.1 测试数据 & 可视化

首先我们运行 Lab03-01 和 Lab03-02 的服务器端和客户端，通过控制我们的延时为 0ms，分别使用丢包率为 0%,20%,40%,60%,80% 进行分别的测试，最后得到如下数据：

表 1: 不同丢包率下的传输时长对比:Time-and-Wait&GBN

Method/Packet Loss	0%	20%	40%	60%	80%
Time-and-Wait	860	5762	18279	32671	64763
GBN	2071	2216	2781	2903	3177

表 2: 不同丢包率下的吞吐率对比:Time-and-Wait&GBN

Method/Packet Loss	0%	20%	40%	60%	80%
Time-and-Wait	6858.83	1023.76	307.91	187.75	89.97
GBN	2897.13	2571.97	2219.04	2198.57	2078.16

我们根据上述的数据，使用 python 对其进行可视化分析。结果如下所示：

传输时间对比：

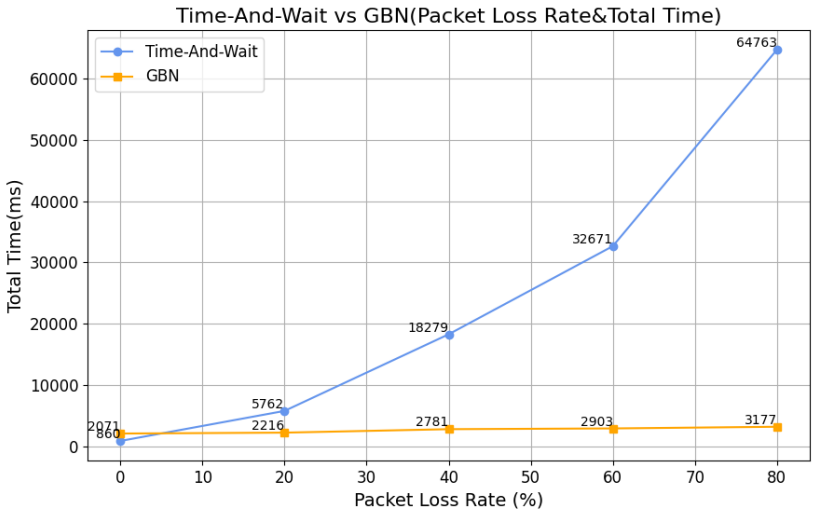

图 4.1: packet-loss-rate&total_time

吞吐率对比：

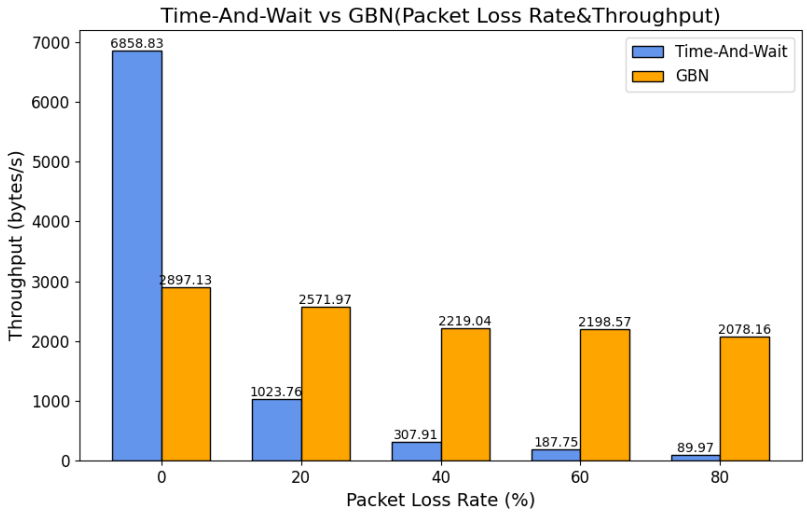

图 4.2: packet-loss-rate&throughput

4.2.2 分析 & 总结

我们从图中可以看出，当丢包率为 0 的时候，是一个特殊的情况。此时**滑动窗口机制的传输时间要比停等机制的要大一些**。理论上来说，当丢包率为 0 的时候，这两者的传输效率应该是差不多的，但是在我们的测试结果中居然相差了足足两倍多，这是为什么呢?

我猜测可能是我在 Lab03-02 中添加了多线程机制，对于日志的输出添加了锁机制，所以在我们的日志进行输出时，会减慢许多，而我们的 Lab03-01 就不一样了，由于是刚开始写的，所以实现较为简单，也没有很多的日志输出来帮我们检查，所以初步认为是 Lab03-02 的多进程和日志锁导致了此处 GBN 的性能比停等机制要来的差。

然后我们继续分析。当丢包率不断增大的时候，我们发现，停等机制的传输效率大幅度下降，从原来的 6800bytes/s 的吞吐率下降到了 89.97bytes/s! 而我们的滑动窗口机制就相对来说十分稳定了，基本上在丢包率为 5% 的时候，我们的 GBN 机制就以及超过了停等机制的性能。GBN 机制性能下降的主要原因就是，在丢包率较大的时候，GBN 机制发生重传的几率也会越来越大，一旦发生重传就会对我们的传输效率产生一些影响，所以此处随着我们的丢包率不断上升，我们的传输效率也在轻微下降，但是下降幅度确实不是很大，这也证实了我们**在丢包率较大的环境下，选用 GBN 来进行传输，是要远远优于停等机制的**，与理论结论相一致。

4.3 不同延时下的性能对比

4.3.1 测试数据 & 可视化

然后，我们运行 Lab03-01 和 Lab03-02 的服务器端和客户端，通过控制我们的丢包率为 0%，分别使用延时为 0ms,120ms,240ms,360ms,480ms 进行分别的测试，最后得到如下数据：

表 3: 不同延时下的传输时长对比:Time-and-Wait&GBN

Method/Latency	0ms	120ms	240ms	360ms	480ms
Time-and-Wait	860	10971	21093	32175	44091
GBN	2071	17183	36971	51634	74935

表 4: 不同延时下的吞吐率对比:Time-and-Wait&GBN

Method/Latency	0ms	120ms	240ms	360ms	480ms
Time-and-Wait	6858.83	537.65	279.65	183.32	133.78
GBN	2897.13	349.18	162.29	116.2	80.07

我们根据上述的数据，使用 python 对其进行可视化分析。结果如下所示：

传输时间对比：

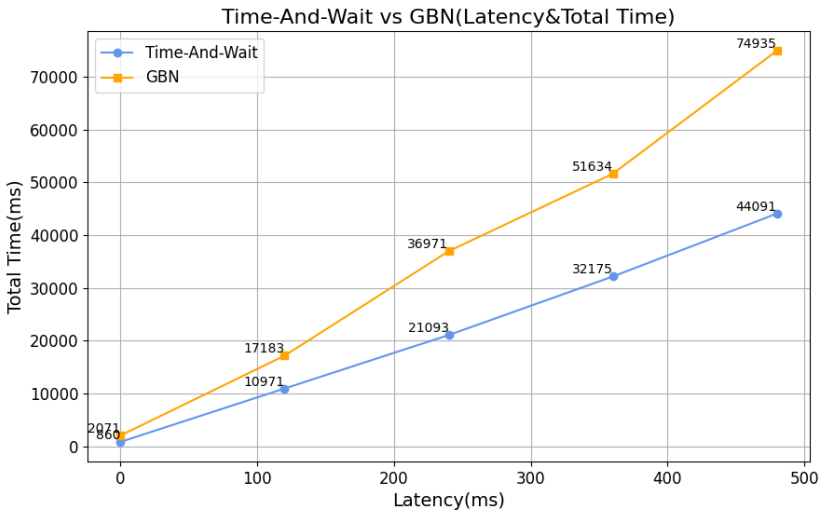

图 4.3: Latency&total_time

吞吐率对比：

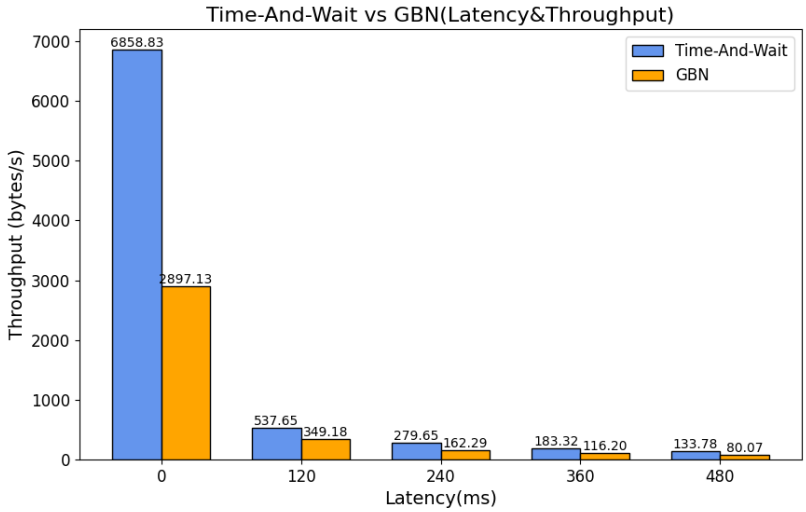

图 4.4: Latency&throughput

4.3.2 分析 & 总结

具体来说，我们可以发现：**在延时传输的情况下，停等机制要优于 GBN 机制，而且两者的传输性能的差距是随着延时的增大而越来越大的！**

首先分析停等机制的传输效率变化。我们在增加的延时条件下，停等机制的传输时间显著增加，且吞吐率持续下降。这是因为在停等机制中，每个数据包的确认都受延时的直接影响，导致整体传输效率下降。但是跟滑动窗口不同的是，延时只会对某一个包产生单独的影响，而不会因为延时而导致全体重传。

相比于停等机制，我们的滑动窗口 GBN 机制的性能下降更为明显。由于 GBN 协议在超时重传情况下和丢包的情况下，同样需要重传整个窗口中的所有数据，所以导致了滑动窗口 GBN 机制的吞吐率急速下降，延时的增加加剧了重传的影响，导致传输时间增加和吞吐率降低，慢于停等协议。

5 滑动窗口机制中不同窗口大小性能对比

5.1 实验设置

本组实验主要用来测试 Lab03-02 中的滑动窗口的大小变化的性能对比，我们还是主要通过改变其丢包率和延时情况来进行测试。

首先，我们统一一下本组实验的测试标准。本组实验，我们还是采用测试文件 2.png 来进行测试，对于 GBN 滑动窗口机制中的窗口大小的不同来进行对比。**首先，我们控制丢包率和延时率均为 0，来测试正常传输的情况下，滑动窗口的大小对传输速率的影响；然后，我们还是按照前面的方法，控制延时为 0ms，测试在不同丢包率 (梯度设置为 0%,20%,40%,60%,80%) 下的传输性能；然后我们控制丢包率为 0，测试在不同延时 (梯度设置为 0ms,120ms,240ms,360ms,480ms) 下的传输性能**。

另外，需要注意的是，由于助教在检查时，要求的窗口大小的范围为 20-32，本次测试中，为了找出普适的规律，**我将滑动窗口大小的范围调整到了 4-32**。

5.2 正常传输下的性能对比

5.2.1 测试数据 & 可视化

首先，我们运行 Lab03-02 的服务器端和客户端，将我们的丢包率和传输时延都设置为 0，然后通过调整滑动窗口的大小，来进行对比测试，通过测试得到如下的数据：

表 5: 正常传输下的性能对比: 传输时长 & 吞吐率

total-time&throughput/windows size	4	8	16	24	32
total-time	1642	2054	2013	1937	2071
throughput	3654.05	2921.11	2980.6	3097.55	2897.13

我们根据上述的数据，使用 python 对其进行可视化分析。结果如下所示：

传输时长 & 吞吐率：

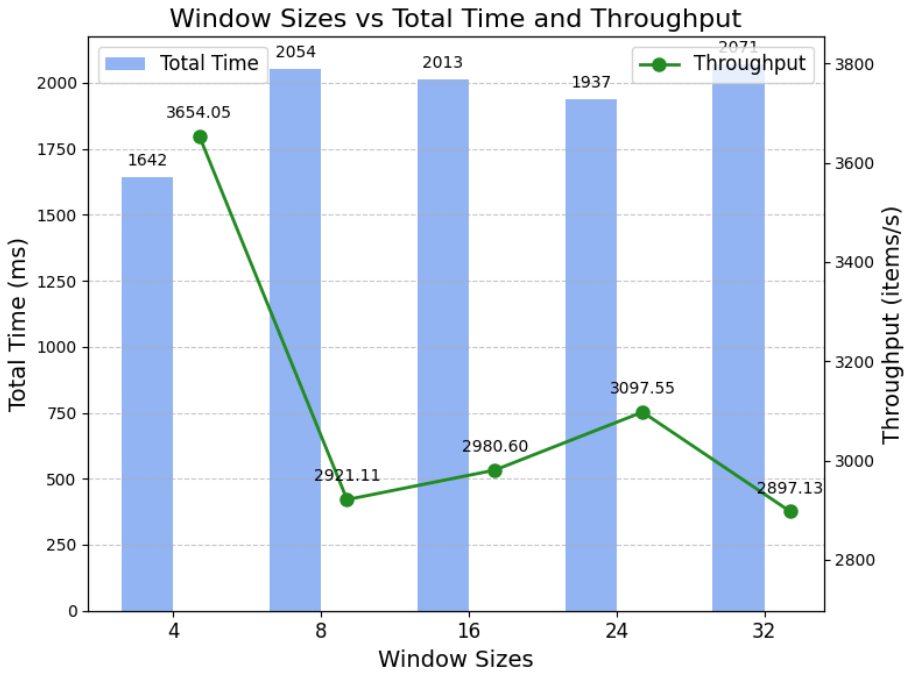

图 5.5: normal-condition_total-time&throughput

5.2.2 分析 & 总结

根据上面的分析，我们可以发现，实际上在正常传输，也就是丢包率和延时都很低甚至为 0 的时候，我们的窗口大小实际上不会对我们的传输性能产生很大的影响，但是我们此处测试发现，在窗口大小较小的时候，我们的传输效率居然是最快的！而我们的窗口大小逐渐增加的过程中，我们发现我们的传输效率实际上越来越低了。

我初步认为，出现这种情况的原因还是我在 Lab03-02 中实现了多线程和日志的输出修改，所以导致在我的窗口大小较大时，**我们的日志输出过多，导致一定时间的浪费**。而在滑动窗口较小的时候，我们不需要输出很多的日志，重传的时候也只需要传输很少的数据报，这样就可以保证我们的传输速率。

5.3 不同丢包率下的性能对比

5.3.1 测试数据 & 可视化

首先我们运行 Lab03-02 的服务器端和客户端，**通过控制我们的延时为 0ms，分别使用滑动窗口大小为 4、16、32，分别使用丢包率为 0%,20%,40%,60%,80%** 进行分别的测试，最后得到如下数据：

丢包率 (%)	窗口大小 4	窗口大小 16	窗口大小 32
0	1642	2013	2071
20	3317	2275	2174
40	4792	2431	2217
60	5148	2754	2364
80	5397	3157	2497

表 6: 不同窗口大小下的传输时间（ms）

丢包率 (%)	窗口大小 4	窗口大小 16	窗口大小 32
0	3654.05	2980.6	2897.13
20	1808.85	2637.33	2759.87
40	1252.08	2468.09	2706.34
60	1165.49	2178.63	2538.05
80	1111.72	1900.52	2402.87

表 7: 不同窗口大小下的吞吐率（bytes/s）

我们根据上述的数据，使用 python 对其进行可视化分析。结果如下所示：

传输时间对比：

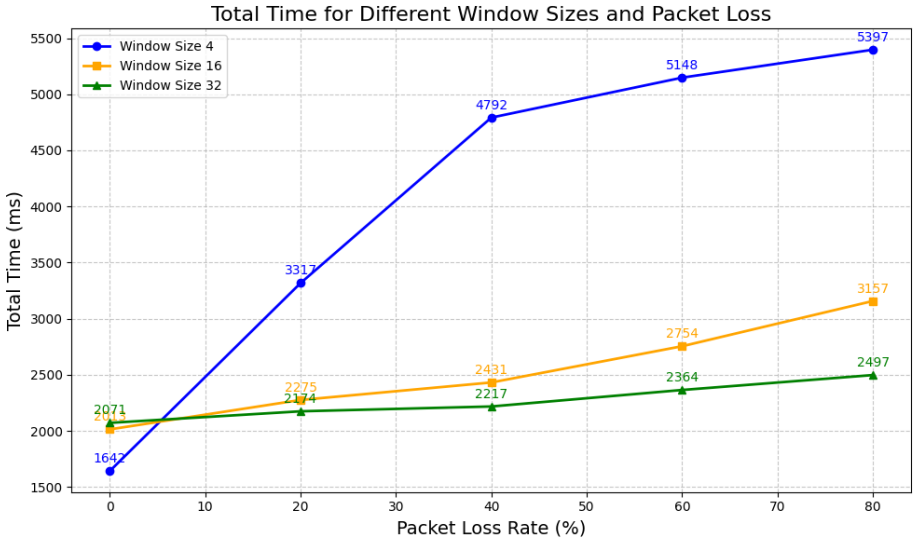

图 5.6: GBN-window-size_total-time(pack-loss)

吞吐率对比：

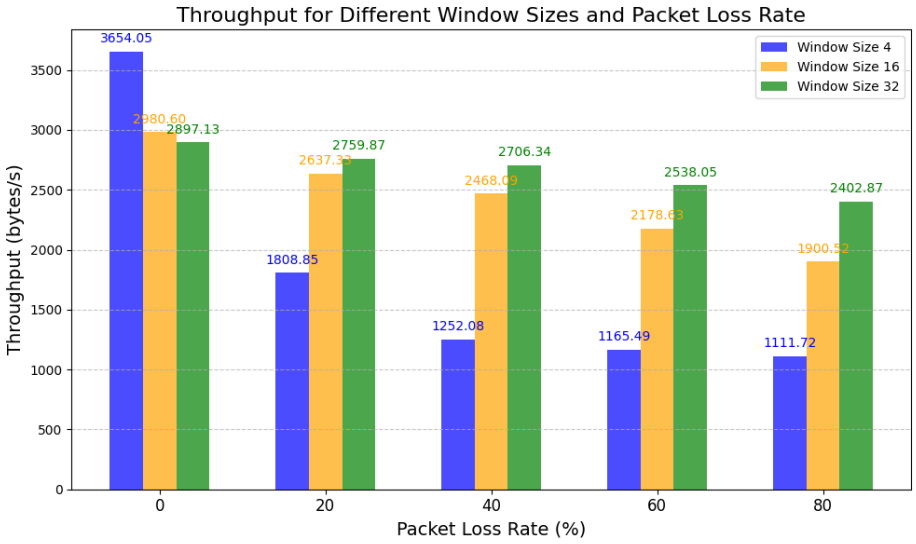

图 5.7: GBN-window-size_throughput(pack-loss)

5.3.2 分析 & 总结

首先，我们可以发现，随着丢包率的上升，我们的几个窗口的大小所对应的传输时间都在上升，对应的吞吐率都在下降。

进一步来看，我发现，在滑动窗口大小较小的时候，对应的传输时间会随着丢包率的上升而迅速上升，波动较大。比如我们的滑动窗口大小为 4 的时候，随着丢包率从 0 到 40% 的上升，我们的传输时间瞬间上升了 3 倍！

而对于窗口较大的情况下，比如 32，我们的丢包率就不会对传输时间有这么大的影响了，仅仅发生一些小的波动。这是因为，**滑动窗口增大后，一次性可以发送更多的数据包，减少了发送次数，因此增大吞吐率**。但是如果丢包率过高的时候，会频繁出现重传的情况，所以也会需要重新传输更多的数据报，这两者是互相制约的，根据我们的结果就可以发现，实际上还是一次发送多个数据报对我们实验的结果影响更大一些。

5.4 不同延时下的性能对比

5.4.1 测试数据 & 可视化

首先我们运行 Lab03-02 的服务器端和客户端，**通过控制我们的丢包率为 0，分别使用滑动窗口大小为 4、16、32，分别使用延时为 0ms,120ms,240ms,360ms,480ms** 进行分别的测试，最后得到如下数据：

Delay (ms)	Window Size 4	Window Size 16	Window Size 32
0	1642	2013	2071
120	17540	16471	15943
240	33784	30714	28941
360	37481	36047	35410
480	41097	40977	39871

表 8: 不同窗口大小下的传输时间（ms）

Delay (ms)	Window Size 4	Window Size 16	Window Size 32
0	3654.05	2980.6	2897.13
120	342.07	364.27	376.34
240	177.59	195.35	207.32
360	160.08	166.45	169.44
480	145.99	146.42	150.48

表 9: 不同窗口大小下的吞吐率（bytes/s）

我们根据上述的数据，使用 python 对其进行可视化分析。结果如下所示：

传输时间对比：

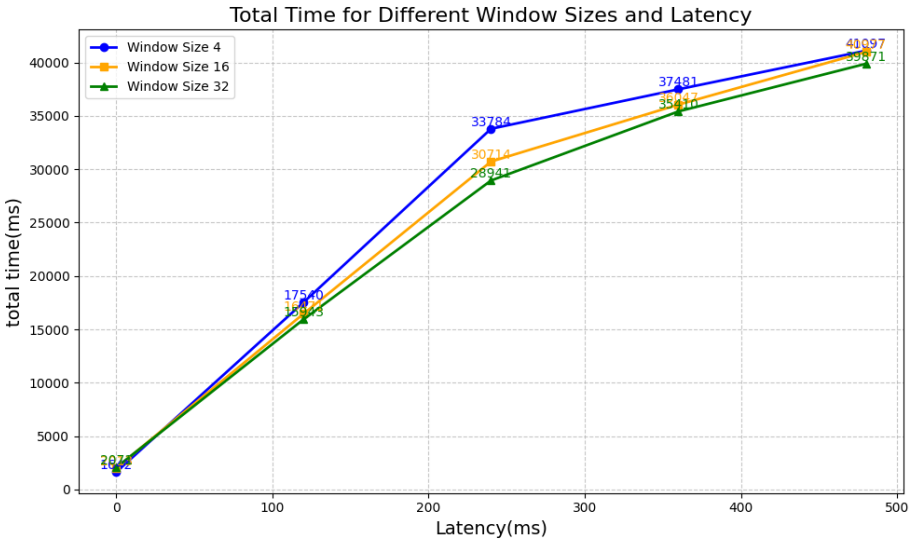

图 5.8: GBN-window-size_total-time(Latency)

吞吐率对比：

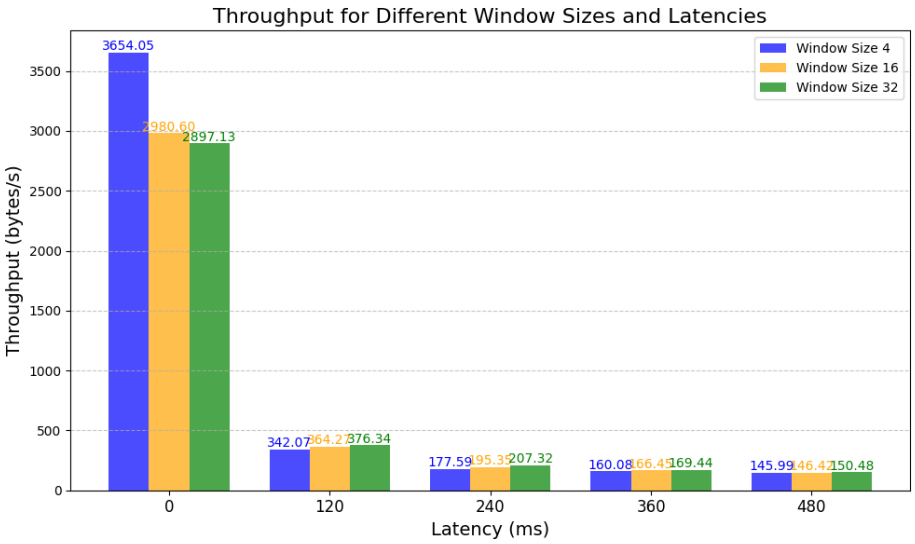

图 5.9: GBN-window-size_throughput(Latency)

5.4.2 分析 & 总结

我们发现，随着时延的上升，我们的传输性能受到了极大的影响。对于我们设定的几个窗口，我们都发现了显著的性能下降。大量的超时重传，导致三个窗口大小范围的 GBN 都会因为大量的重传而性能不断恶化。并且我们设定的三个窗口的数据差异不大，都处于比较低水平。**所以，窗口大小即使增大，那些多发的数据报也没有办法使传输的性能得到提升，因为我们的延时对性能的影响实在是太大了。**

6 有拥塞控制 & 无拥塞控制性能对比

6.1 实验设置

本组实验主要用来测试 Lab03-02 与 Lab03-03 的有无拥塞控制的变化性能对比，我们还是主要通过控制丢包率和延时情况来进行测试。

首先，我们统一一下本组实验的测试标准。我们还是采用测试文件 2.png 来进行测试，对于有无拥塞控制的算法，分别测试改变丢包率和时延的大小，梯度和前面的几组实验相同，不再说明，然后去记录我们的有无拥塞控制对传输效率产生的影响即可。我们设置我们的滑动窗口大小为 32。

需要注意的是，在 Lab03-03 的编写中，**我们使用了更多的输出日志，来确保我们的拥塞控制的正确性，因此我们多余的日志部分会在一定程度上影响我们的传输效率**，所以我们的结果会有一些与理论冲突。

6.2 不同丢包率下的性能对比

6.2.1 测试数据 & 可视化

首先，我们运行 Lab03-02 和 Lab03-03 的服务器端和客户端，将我们的时延调整为 0，将丢包率初始化为 0。然后，我们通过调整我们的丢包率来进行对比测试，通过测试可以得到如下的数据：

Packet Loss Rate (%)	No Congestion	With Congestion
0	2071	2107
20	2216	2201
40	2781	2304
60	2903	2417
80	3177	2531

表 10: 不同丢包率下的性能对比 (传输时间)

Packet Loss Rate (%)	No Congestion	With Congestion
0	2897.13	2847.63
20	2571.97	2726.01
40	2219.04	2604.15
60	2198.57	2482.39
80	2078.16	2370.59

表 11: 不同丢包率下的性能对比 (吞吐率)

我们根据上述的数据，使用 python 对其进行可视化分析。结果如下所示：

传输时间对比：

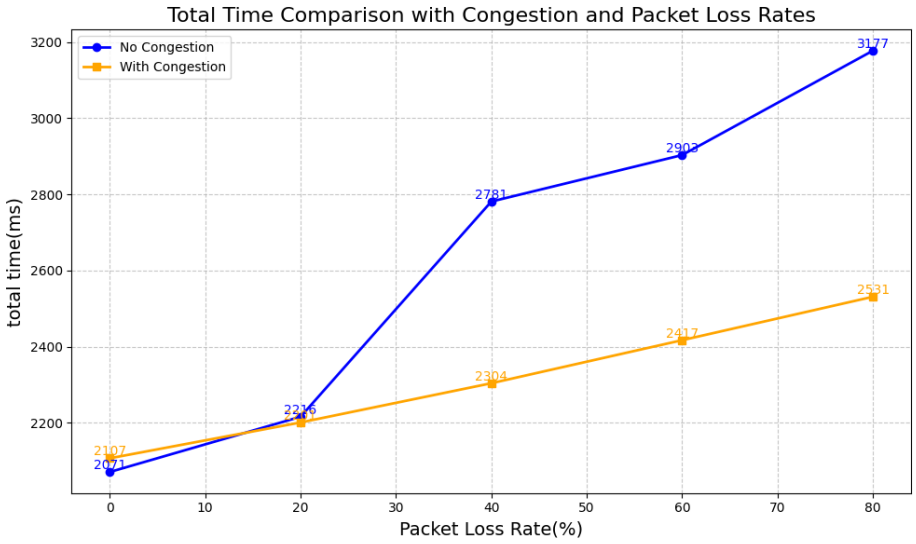

图 6.10: Congestion_total-Time(pack-loss)

吞吐率对比：

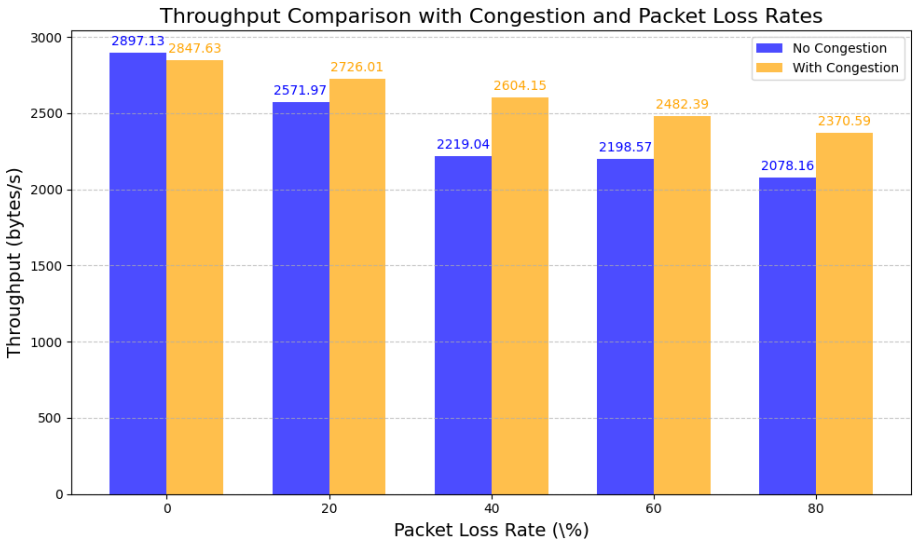

图 6.11: Congestion_Throughput(pack-loss)

6.2.2 分析 & 总结

我们可以发现，在我们时延和丢包率都为 0 的时候，为正常传输，我们的无拥塞控制传输速率是要比拥塞控制传输快一些的，具体原因就是因为，我们的拥塞控制虽然不需要进行重传，但是我们需要经历一次慢启动的过程，所以相比来说我们的传输速率会略慢于我们的无拥塞控制。

然后，当我们的丢包率上升的时候，我们发现，拥塞控制的性能实际上下降地比较慢，而我们的无拥塞控制受到了很大的影响。原因可能是，由于多次重传，导致我们的窗口内的所有数据都需要重新传输，**而我们的拥塞控制中的 RENO 机制通过慢启动和拥塞避免自动下调我们的窗口大小，很好地解决了丢包过多而导致性能下降的问题**。可以看到，拥塞控制的性能在丢包率为 20% 的时候就已经超过了无拥塞控制，实际上可能在 10% 左右的丢包率，也是这样的情况。

6.3 不同延时下的性能对比

6.3.1 测试数据 & 可视化

首先，我们运行 Lab03-02 和 Lab03-03 的服务器端和客户端，将我们的丢包率调整为 0，将时延初始化为 0。然后，我们通过调整我们的时延来进行对比测试，通过测试可以得到如下的数据：

Latency (ms)	No Congestion	With Congestion
0	2071	2107
120	7671	2471
240	12703	2897
360	16513	3261
480	22971	3719

表 12: 不同延时下的性能对比 (传输时间)

Delay (ms)	No Congestion	With Congestion
0	2897.13	2847.63
120	782.20	2428.15
240	472.35	2071.09
360	363.35	1839.91
480	261.19	1613.33

表 13: 不同延时下的性能对比 (吞吐率)

我们根据上述的数据，使用 python 对其进行可视化分析。结果如下所示：

传输时间对比：

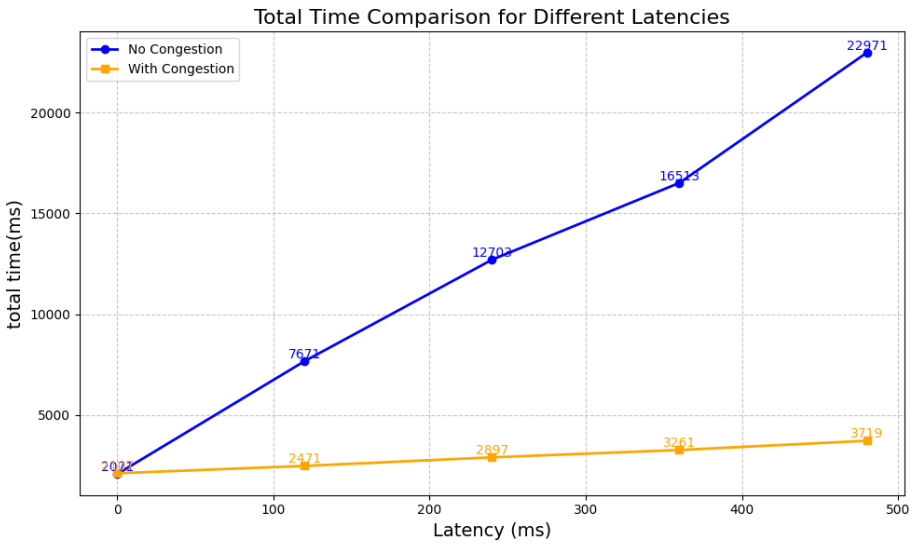

图 6.12: Congestion_total-Time(Latency)

吞吐率对比：

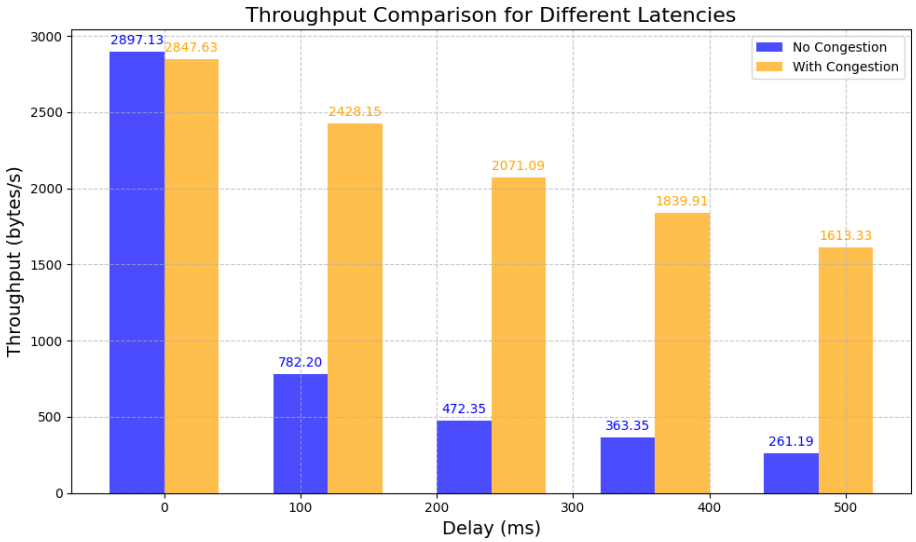

图 6.13: Congestion_Throughput(Latency)

6.3.2 分析 & 总结

我们同样发现，改变时延所导致的结果和前面的改变丢包率的结果大体类似。**当时延较低时，无拥塞控制与有拥塞控制性能相当；当时延较高时，拥塞控制的性能比无拥塞控制的性能要更好一些**。

有拥塞控制时，时延较大的时候，**RENO 算法会执行快速重传的操作，使我们的传输性能得到提升，并且还会自动下调窗口的大小**，避免超时后需要重传的过多。而我们的无拥塞控制就会在较多的时延中全部将窗口内的数据进行重传，大大影响了我们传输文件的性能。

7 心得体会

本学期共完成了 6 个计算机网络的实验。第一个实验为后续的 C/S 传输打下了基础，第二个实验学习和使用了 wireshark 的一些相关知识。第三个实验分为四个小实验，分别完成了停等机制、滑动窗口机制以及拥塞控制的相关设计。

此处主要说一说我在 Lab03-04 中的收获与思考。对于实验数据不能与理论完全符合时，我会通过我编写的代码去寻找问题所在，最后对我的实验数据给出一个很好的解释。

临近期末，学业压力较大，所以我对于本次实验仍然有一些未完全优化的地方，比如没有解决日志进程拖慢传输进度导致测量不精确的问题。

通过一学期的计算机网络的编程训练，我对 socket 编程已经有了很好的理解，并且能够自己去设计一些协议，并通过代码来进行实现。

感谢两位老师理论课的讲授；感谢助教学长平时的点拨，也感谢自己，在忙碌的大三上学期坚持了下来，完成了《计算机网络》本学期最后一个实验。